

Data and Computation Modeling for Scientific Problem Solving Environments

Alex A. Verstak

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE
in
Computer Science and Applications

Dr. Naren Ramakrishnan, Chair _____

Dr. Layne T. Watson _____

Dr. William H. Tranter _____

August 15, 2002
Blacksburg, Virginia

Keywords: Problem solving environments, wireless system design, experiment management,
data mining, wireless propagation prediction.

Copyright 2002, Alex A. Verstak

Data and Computation Modeling for Scientific Problem Solving Environments

Alex A. Verstak

Abstract

This thesis investigates several issues in data and computation modeling for scientific problem solving environments (PSEs). A PSE is viewed as a software system that provides (i) a library of simulation components, (ii) experiment management, (iii) reasoning about simulations and data, and (iv) problem solving abstractions. Three specific ideas, in functionalities (ii)-(iv), form the contributions of this thesis. These include the EMDAG system for experiment management, the BSML markup language for data interchange, and the use of data mining for conducting non-trivial parameter studies. This work emphasizes data modeling and management, two important aspects that have been largely neglected in modern PSE research. All studies are performed in the context of S⁴W, a sophisticated PSE for wireless system design.

Acknowledgments

The work presented here would have been impossible without the assistance of many individuals.

First and foremost, I would like to acknowledge the enormous contribution of Christopher Anderson. He patiently taught me a number of wireless system concepts that were hard to grasp for a person with background in computer science. Chris also provided measurements data for ray tracer calibration.

Dr. Naren Ramakrishnan, my advisor, was a tremendous help in reviewing the thesis and providing motivation for the tedious writing process. Dr. Ramakrishnan's comments have greatly improved the organization and the clarity of the material. His fabulous ability to find the right resources at the right time provided an excellent counter-weight to my total lack of time management skills.

Dr. Layne T. Watson was extremely helpful in achieving mathematical correctness and succinctness of the presentation. His high demands for the quality of the presentation and tireless bashing of non-standard notation have uncovered a large number of logic flaws and blind spots. I am grateful to Dr. Watson that these flaws have not made it beyond the early drafts of the thesis.

I am also thankful to Dr. William H. Tranter for his invaluable and timely feedback on Chapter 4 of this thesis.

Finally, I would like to acknowledge the hard work of Kyung K. Bae and Jian He in developing the WCDMA simulation and the VTDIRECT numerical optimizer. Most of this thesis deals with the integration issues that would be futile without the software to integrate.

Contents

1	Introduction	1
2	Experiment Management	3
2.1	EMDAG	3
2.1.1	Installation Management	6
2.1.2	Execution Management	9
2.1.3	Data Management	14
2.2	Computation and Data Models in S ⁴ W	16
2.2.1	Ray Tracing Model and Data Management	16
2.2.2	WCDMA Simulation Model and Data Management	25
2.2.3	Numerical Optimization and Data Management	29
2.3	Discussion of Experiment Management	33
3	Data Interchange with BSML	34
3.1	Motivating Example	34
3.2	PSE Requirements for Data Interchange	37
3.2.1	Related Work	37
3.3	Validation	39
3.4	Binding	45
3.5	Conversion	50
3.6	Integration with a PSE	58
3.6.1	Status of Prototype	58
3.7	Discussion of BSML	59
4	Data Mining Study	61
4.1	In this Chapter	62
4.2	The Statistics of Aggregation and the Aggregation of Statistics	62
4.2.1	The First Level of Aggregation: Points	64
4.2.2	The Second Level of Aggregation: Buckets	64
4.2.3	Confidence Estimation	65
4.3	Extended Example	66
4.3.1	Statistically Significant Sampling Methodology	70
4.3.2	Results of Statistically Significant Sampling	70
4.4	The Third Level of Aggregation: Regions	74
4.4.1	Region Shape	76
4.4.2	Region ‘Goodness’	77

4.4.3	Optimized Regions	79
4.5	Optimized-Support Regions for the STTD Example	81
4.5.1	Justification of Data Mining for the STTD Example	81
4.5.2	Optimized-Support Admissible Regions	82
4.6	Discussion and Future Work	85
A	EMDAG Schemas	91
A.1	Installation Management Schemas	91
A.2	Execution Management Schemas	96
B	S⁴W Schemas	115
B.1	S ⁴ WRT Schemas	115
B.2	WCDMA Simulation Schemas	125
B.3	VTDIRECT Schemas	127
C	BSML DTD	136

List of Figures

2.1	Summary of schema notation.	5
2.2	Overview of experiment management schemas.	5
2.3	Installation management schemas.	6
2.4	WCDMA simulation startup.	9
2.5	Execution management schemas.	10
2.6	Data management schemas.	15
2.7	2D beam tracing.	17
2.8	Beam intersection with a receiver location grid.	18
2.9	Environment preprocessing schemas.	22
2.10	Ray tracing schemas.	23
2.11	Block diagram of the WCDMA simulation.	26
2.12	WCDMA simulation schemas.	27
2.13	VTDIRECT schemas.	30
3.1	A site-specific system model in S^4W	35
3.2	Transmitter placement optimization.	36
3.3	XML schemas for an octree environment decomposition.	41
3.4	Non-XML schemas for an octree environment decomposition.	42
3.5	LL(1) grammar corresponding to the octree schemas in Figures 3.3 and 3.4.	43
3.6	L-attributed definition of BSML.	44
3.7	Binding schema and parsing tables for a power delay profile.	47
3.8	XML and Matlab versions of a PDP.	48
3.9	Two slightly different schemas for a collection of antennas.	52
3.10	Conversion schema for Figure 3.9.	53
3.11	'Determines' relation.	54
3.12	BSML integration with PSE execution environment.	59
4.1	Typical 1D slices of the configuration space.	63
4.2	BEP estimates for a space of configurations.	67
4.3	1D slices of the surface in Figure 4.2.	68
4.4	A surface fitted onto the data in Figure 4.2.	69
4.5	Empirical CDF for one of the configurations.	71
4.6	Sample sizes for Figure 4.2.	72
4.7	Sample standard-deviation-to-mean ratios for Figure 4.2.	73
4.8	Probabilities that configurations in Figure 4.2 exhibit acceptable performance.	75
4.9	Types of admissible regions.	76
4.10	Why optimal regions are admissible.	82

4.11	Optimized-support admissible regions for Figure 4.2.	83
4.12	Cross-validation of optimized-support admissible regions.	84

List of Tables

2.1	Tcl scripts for interfacing with computing systems.	8
2.2	Installation management database functions.	8
2.3	Execution and installation management functions.	11
2.4	SQL functions for ‘raw’ impulse responses.	24
2.5	SQL functions for ‘sampled’ impulse responses.	27
3.1	A survey of PSE-like systems and XML technologies.	39
4.1	Summary of mathematical notation.	64
4.2	Summary of region notation.	74

Chapter 1

Introduction

Problem solving environments (PSEs) are high-level software systems for doing computational science. A simple example of a PSE is the Web PELLPACK system [40] that addresses the domain of partial differential equations (PDEs). Web PELLPACK allows the scientist to access the system through a Web browser, define PDE problems, choose and configure solution strategies, manage appropriate hardware resources (for solving the PDE), and visualize and analyze the results. The scientist thus communicates with the PSE in the vernacular of the problem, ‘not in the language of a particular operating system, programming language, or network protocol’ [28]. It is 10 years since the goal of creating PSEs was articulated by an NSF workshop (see [28] for findings and recommendations). From providing high-level programming interfaces for widely used software libraries [46], PSEs have now expanded to diverse application domains such as wood-based composites design [31], aircraft design [30], gas turbine dynamics simulation [22], and microarray bioinformatics [6].

The basic functionalities expected of a PSE include supporting the specification, monitoring, and coordination of extended problem solving tasks. Many PSE system designs employ the *compositional modeling* paradigm, where the scientist describes data-flow relationships between codes in terms of a graphical network and the PSE manages the details of composing the application represented by the network. Compositional modeling is not restricted to such model specification and execution but can also be used as an aid in performance modeling of scientific codes [2] (model analysis).

In this thesis, we view a PSE as providing the following functionalities, in increasing order of sophistication:

1. a library of simulation components,
2. experiment management,
3. reasoning about simulations and data, and
4. problem solving abstractions.

The first item is well recognized as an integrated aspect of a PSE [46]. Simulation components are domain-specific software modules that play the most important role in defining the PSE. These modules are composed to achieve the desired functionality. The second item (experiment management) refers to management of simulation execution and data. Execution management involves scheduling and monitoring the status of simulation runs as well as composing the simulation components into given interconnection topologies. Data management is concerned with recording and analyzing simulation data. The primary contributions of this research are (i) the integration of data management and execution management by the means of a relational database and (ii) the experimental verification of this integration methodology via data models for a particular PSE. Item 3 (reasoning) is the next level of functionality above data management. Reasoning

can be viewed as modeling and analyzing metadata, i.e., the data about simulation components and about simulation data. One example of reasoning functionality is automatic generation of conversion filters in Chapter 3. In general, reasoning is concerned with using a domain theory to support inference about PSE models and data. Finally, problem solving abstractions (item 4) are built upon the lower-level functionalities. We use this term to refer to numerical optimization [39], data mining [23], and recommendation [44], which are currently the highest level of functionality available in PSEs.

This thesis studies the generic PSE issues above in the context of S⁴W (Site-Specific System Simulator for Wireless system design). S⁴W is a PSE being developed at Virginia Tech [50]. This PSE provides deterministic electromagnetic propagation and stochastic wireless system models for predicting the performance of wireless systems in specific environments, such as office buildings. S⁴W is also designed to support the inclusion of new models into the system, visualization of results produced by the models, integration of optimization loops around the models, validation of models by comparison with field measurements, and management of the results produced by a large series of experiments. The complexity and sophistication of S⁴W make it an appropriate vehicle to motivate and demonstrate the ideas developed in this thesis.

This thesis is organized as follows. Chapter 2 presents EMDAG, a generic experiment management system. EMDAG is implemented on top of PostgreSQL, an advanced and freely available relational database management system. This chapter also serves as an overview of the S⁴W components and data models. We emphasize the integration between a ray tracing propagation model, a WCDMA (wideband code division multiple access) wireless system model, and a numerical optimizer. However, the individual software components are not covered in any detail. Chapter 3 switches from the relational to the semistructured data model. The material in this chapter covers both the execution management (binding) and the reasoning (automatic format conversion) functionalities. This chapter combines ideas from markup languages, semistructured data, and change detection. Chapter 4 deals with data mining, which is an abstraction at the highest level of PSE functionality. We explore a hierarchical approach to aggregation of wireless system performance metrics. A well known region mining algorithm is specialized to take problem-specific information into account. Given a number of the WCDMA simulation runs, this algorithm finds optimal-performance regions in the space of wireless system configurations. Finally, the appendices contain detailed schemas for the various data models presented in the thesis.

Each of the three chapters that follow is more or less self-contained. All of these chapters deal with substantially different PSE functionalities. Thus, each chapter has its own introduction and its own summary.

A good problem solving facility is ultimately characterized by ‘what it lets you get away with.’ The systems described in Chapters 2 and 3 allow model engineers to flexibly incorporate application-specific considerations for data management and data interchange, without insisting on an implementation vocabulary for the components. The data mining methodology developed in Chapter 4 permits researchers to conduct non-trivial parameter studies with a relatively mild qualitative model of the underlying simulation.

Chapter 2

Experiment Management

This chapter presents management of scientific experiments as an exercise in database modeling and management. A scientific experiment consists of data preparation, simulation or measurements, and data analysis. These three phases of experiment execution can be repeated and interleaved in arbitrary ways. In database terms, simulation corresponds to loading data into the database, data analysis corresponds to querying the database, and data preparation corresponds to a combination of loading and querying.

The database view of experiment management has been explored in many projects. The ZOO [36] desktop experiment management environment aims to achieve goals similar to ours with an object oriented wrapper around a relational database. However, ZOO is too generic to be directly applicable in a PSE setting. In particular, it does not integrate with execution of parallel simulations, a task common in S⁴W. The Tioga2 [4] scientific visualization environment provides data visualization tools on top of a relational database. This work is largely orthogonal to the system presented here. Software similar to Tioga2 could be used to visualize simulation data accumulated in our experiment database. Tioga2 does not integrate with simulation execution either.

Many systems that *do* integrate with simulation execution, however, do not provide a database interface. Examples of such systems are Globus and Legion [24]. Systems of this type provide state-of-the-art parallel job scheduling and computational resource allocation facilities. However, such facilities operate at a relatively low-level compared to the needs of a PSE.

This chapter develops a hybrid approach that incorporates both simulation data management and simulation execution management. Section 2.1 describes EMDAG, a generic experiment management system, and Section 2.2 describes S⁴W, a specific application of this system. This description of S⁴W consists of three parts: the ray tracing propagation model, the Monte Carlo WCDMA wireless system model, and the DIRECT numerical optimizer. This chapter emphasizes data modeling and integration issues and shows how these issues are addressed by a relational database. We demonstrate that data management can be usefully harnessed, in multiple ways, in a PSE.

2.1 EMDAG: An Experiment Manager for Data Acquisition and Generation

This section describes EMDAG—a generic system for management of scientific experiments. EMDAG manages execution of computationally expensive simulations on a variety of computing systems and facilitates storage of simulation results in a relational database. EMDAG is designed for simulations whose computational expense makes database queries over stored simulation results faster than repeated execu-

tion of simulations. EMDAG's approach to experiment management is a middle ground between the data-centric approach (ZOO, Tioga) and the computation-centric approach (Globus, Legion). The contribution of EMDAG is integration of both data-centric and computation-centric philosophies into a single coherent system. A working prototype of this system is the backbone of S⁴W.

EMDAG is implemented as (a) a set of tables, functions, and triggers in a relational database (PostgreSQL), (b) a number of Tcl scripts for interfacing with computational systems, and (c) a Tcl package to facilitate development of new simulation wrappers. Database tables contain catalogs of simulation software, computing systems, and simulation data, as well as up-to-date status of simulation runs. Database functions can be invoked by database clients to control simulation execution. Database triggers enforce referential integrity of EMDAG's tables and notify database clients of certain simulation execution events. Tcl scripts are used by database functions to start and stop simulations on Unix workstations and clusters of Unix workstations. These scripts merely interface to commonly used job schedulers. It is the responsibility of the simulations or simulation wrappers to read the inputs from the database and write the outputs to the database. Finally, the Tcl package provides a simplified database interface to simulation wrappers. A subset of this package is also available in C.

Although EMDAG can be used to manage measurements data, it is primarily concerned with simulations and simulation results. Three components of experiment management can be identified in this setting: installation management, execution management, and data management. Installation and execution management are computation-centric functionalities while data management is a data-centric functionality. All three functionalities must be provided to conduct a scientific experiment.

Installation and execution management are independent of the simulation software. Similarly to Globus and Legion, EMDAG can manage installation and execution of any set of simulation components. We only require that the components be computationally expensive, i.e., that the overhead of database access be justified by the ability to query simulation results in the future. Data management necessarily depends on the simulations. Similarly to ZOO and Tioga, we defer schema design and database input/output (I/O) for any given simulation to simulation developers. Database I/O can be either built into the simulation or provided by a wrapper script. Examples of schemas and database I/O implementations in S⁴W are given in the next section. Several conventions must be followed to integrate a simulation with EMDAG, but, like any component framework, EMDAG leaves considerable freedom of choice to developers.

Schemas will be our primary tool in explaining EMDAG and experiment management. Schema notation conventions are described in Figure 2.1. The top level view of EMDAG schemas is given in Figure 2.2. This figure shows the relations between data management, execution management, and installation management tables. Let us look at this figure in more detail.

Simulation runs are organized as a collection of experiments. For the purpose of this work, an *experiment* is a number of related runs of a *single* simulation. Composition of *multiple* simulations amounts to cross-referencing experiments (we will see examples of this in the next section). There is one row per experiment in the `experiments` table. Each experiment is identified by a string (`id` column). The `component` and `installation` columns refer to a software installation of the simulation component that executed (or will execute) the experiment. An experiment consists of a number of *points* (runs) that correspond to particular simulation inputs. Execution states of these points are stored in the `experiment_point_states` table. Simulation inputs are stored in a component-specific `experiment_inputs` table. There will be a separate inputs table for each simulation component (see next section for examples). We use the name `experiment_inputs` to refer to any table of this kind. Experiment points are identified by experiment `id` and a sequence number (`id` and `seq_no` columns).

The following subsections describe installation management, execution management, and data management in more detail. The material is organized bottom-up. The lowest level of functionality (installation

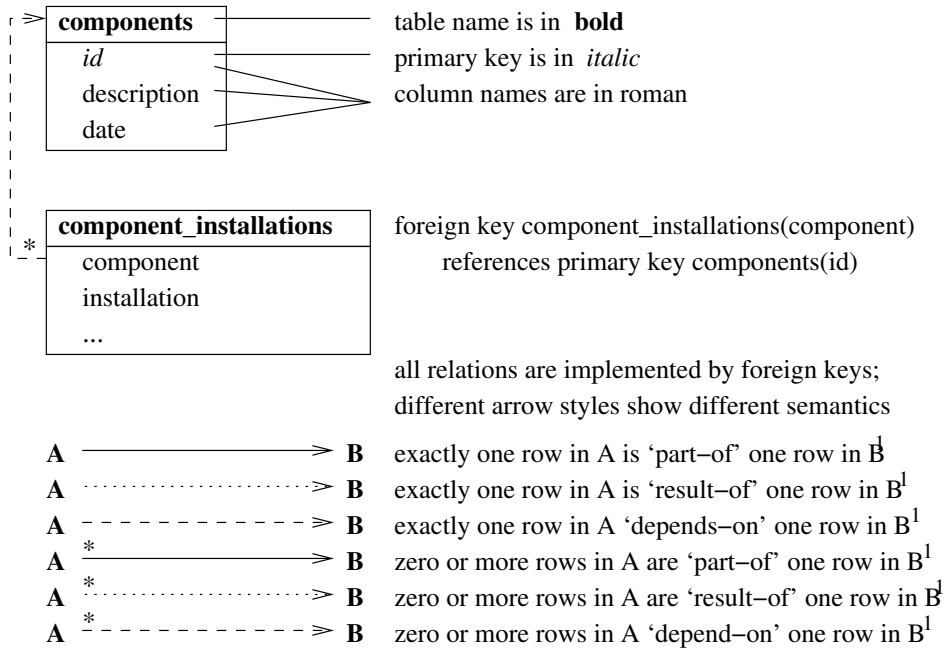


Figure 2.1: Summary of schema notation. This notation is similar (but not identical) to entity/relationship (E/R) diagrams. We use a compact format for entity (table) attributes and different arrow styles for different types of relations.

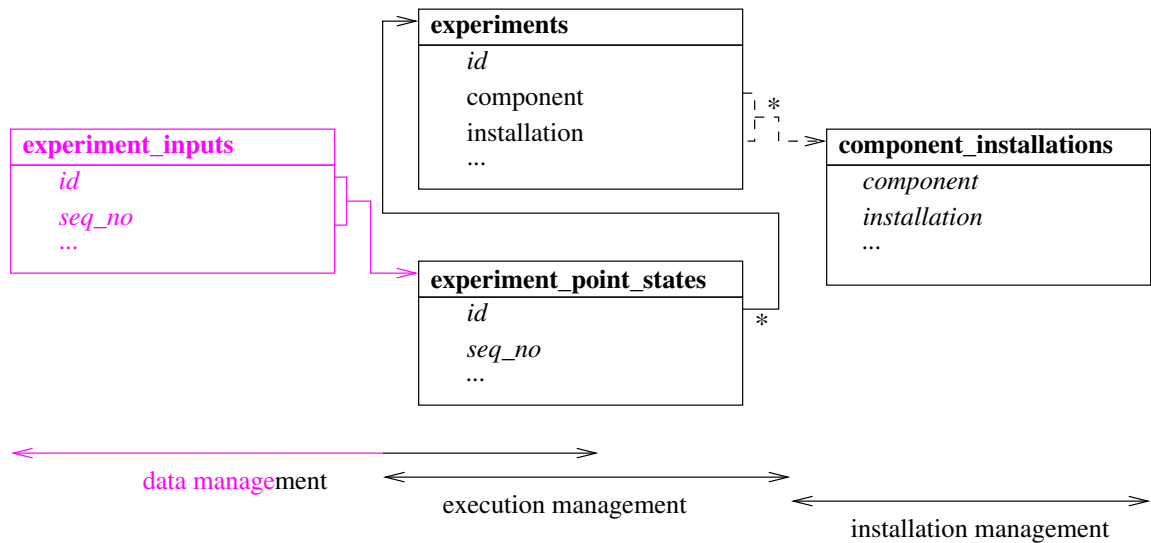


Figure 2.2: Overview of experiment management schemas. SQL code for all schemas is available in Appendix A.

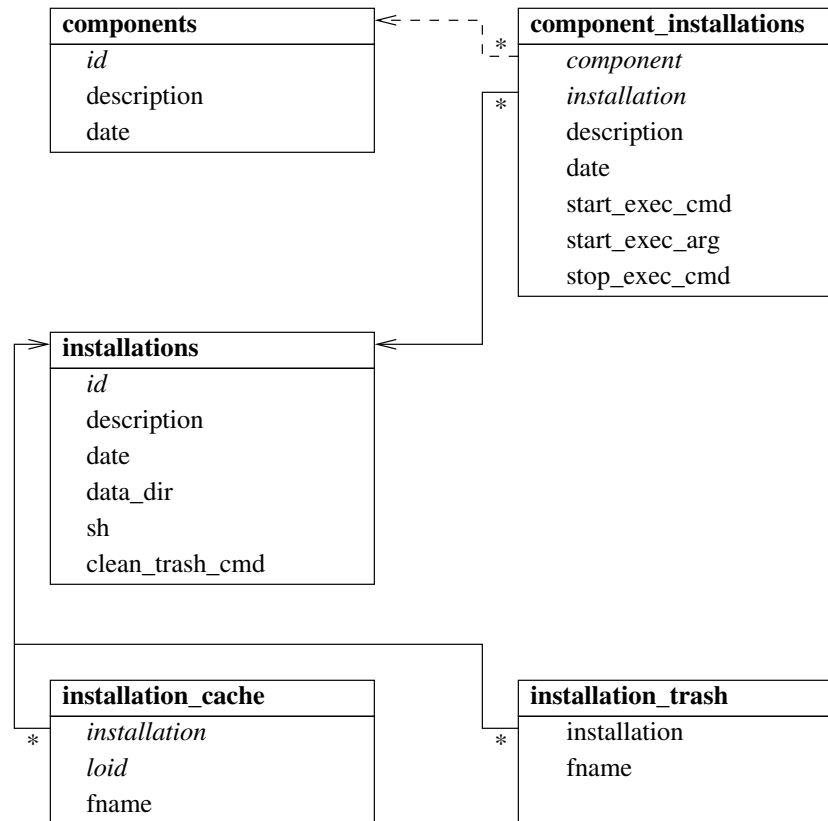


Figure 2.3: Installation management schemas. SQL code for these schemas is available in Appendix A.1.

management) is thus presented first. Then, we build higher-level functionality (execution management) on top of lower-level functionality. The description of EMDAG is concluded with several remarks about the highest level of functionality available in this system (data management). The data management subsection is necessarily brief and abstract. The rest of the chapter is devoted to the instantiation of these functionalities for S⁴W.

2.1.1 Installation Management

Installation management is concerned with installations of simulation components on available computing systems. Installation management tables (Figure 2.3) contain sufficient information to start and stop simulation execution on a given computing system. These tables also facilitate caching of simulation inputs and outputs.

The `installations` table is the heart of installation management. An *installation* is a computing system (a computer or a cluster) where a set of simulation components is installed. EMDAG database functions use the shell (`sh` column) to execute commands on computing systems. Commands to execute for various tasks are given in the corresponding columns of the installation management tables (described below). The shell and commands are stored as Tcl lists of arguments to the `exec ()` function. EMDAG simply concatenates the shell, the appropriate command, and task-specific arguments, and passes the resultant list of arguments to `exec ()`. Thus, the shell can be any command available on the database host to the database

server process. Likewise, various commands can be any legal arguments to the shell.

For example, S⁴W installations at Virginia Tech use Secure Shell for the shell and custom Tcl scripts for the commands. Consider a database on host `arabidopsis.cs.vt.edu` and an installation on the Beowulf cluster whose front end machine is `sioux.cs.vt.edu`. We can let the installation id be `sioux` and the shell be

```
ssh -l s4w -x -o {BatchMode yes} sioux.cs.vt.edu,
```

where `-l s4w` means login as user `s4w`, `-x` means do not forward X Windows traffic, `-o {BatchMode yes}` means do not prompt for the password (curly braces make ‘BatchMode yes’ one command-line argument instead of two), and `sioux.cs.vt.edu` is the host to connect to. Host `arabidopsis` runs the database server process as user `postgres`, so user `postgres` must be able to connect to the cluster (as user `s4w`) via `ssh` without typing in a password (we use DSA public key authentication). All commands in installation management tables will be appended to the shell’s command line, so they must be commands on host `sioux.cs.vt.edu`. The advantages of this setup over systems like Globus and Legion are simplicity of configuration and use of standard software that is usually already installed on computing systems.

The `installation_cache` and `installation_trash` tables deal with caching simulation inputs and outputs. One application of this facility is caching floor plans in S⁴W. Floor plans are stored as (compressed) large objects in the database. Large objects are identified by large object ids (loids) in PostgreSQL. When a large object is needed by a simulation, it is copied from the database into the computing system’s filesystem. This object is then recorded in the `installation_cache` table and retained in the filesystem until explicitly deleted. The `data_dir` column of `installations` is the directory name and the `fname` column of `installation_cache` is the file name of the cached file, and the `loid` column of `installation_cache` is the large object id in the database. When a row is deleted from the `installation_cache` table, a trigger inserts a corresponding row into the `installation_trash` table. The latter table contains the names of the cached files that should be cleaned up. The column `clean_trash_cmd` in `installations` does precisely that. Since connection establishment overhead is high, cached files are not deleted immediately when the records in `installation_cache` are deleted. It is faster to accumulate records in `installation_trash` and then delete all of these files in one batch.

The clean trash command for installation `sioux` on host `arabidopsis.cs.vt.edu` is

```
clean_trash -h arabidopsis.cs.vt.edu -U s4w -D s4w -o,
```

where `clean_trash` is the script on host `sioux.cs.vt.edu`, the `-h`, `-U`, and `-D` options tell the script which database to connect to, and `-o oid` is installation object id (the `oid` value is appended by EMDAG). We pass object ids (oids), not installation ids or filenames, because strings input by users can have special characters that must be escaped in a shell-dependent way. There are good uses for shell’s special characters, but they tend to confuse PSE users most of the time. Since we are not communicating filenames to the script, we must pass enough information for the script to find them. We pass database connection options and installation oid. One subtle implication of this design decision is that the script must be able to connect to the database that invoked it without providing a password.

Performing work outside of the database also implies that transactional semantics of database queries can be violated. In this case, the database function blocks until the script completes execution. Then, this database function (not the script) deletes the records from `installation_trash`. The danger here is the possibility of a script failure while deleting files. The database function will then fail, so some files will be deleted from the filesystem, but the corresponding records will remain in `installation_trash`. This particular violation of transaction atomicity can hardly cause any problems. In general, the implications of breaking transactional semantics must be evaluated for each command.

start command	stop command	description
<code>start_process</code>	<code>stop_process</code>	interface to Unix process scheduler
<code>start_mpirun_job</code>	<code>stop_mpirun_job</code>	interface to mpirun (MPICH, LAM)
<code>start_pbs_job</code>	<code>stop_pbs_job</code>	interface to PBS (portable batch system)
<code>pbs_mpirun</code>		mpirun (MPICH, LAM) under PBS
<code>pbs_mpiexec</code>		mpiexec (native Myrinet) under PBS
<code>start_manual_job</code>	<code>stop_manual_job</code>	place holder for measurements and debugging

Table 2.1: Tcl scripts for interfacing with computing systems. A uniform database interface to these scripts is described in Table 2.2.

example invocation	description
<code>SELECT start_experiment_executor('wcdma_test')</code>	start experiment <code>wcdma_test</code>
<code>SELECT stop_experiment_executor(17)</code>	stop simulation number 17
<code>SELECT clean_trash('sioux')</code>	remove trash files on installation <code>sioux</code>

Table 2.2: Installation management database functions. These functions should not be used in multi-row queries because no rollback mechanism is available. This limitation is mostly due to lack of exception handling in PostgreSQL.

The `components` and `component_installations` tables in Figure 2.3 provide catalogs of simulation components and component installations. The `component_installations` table contains commands to start and stop simulation execution on a given computing system. This table serves as the interface of execution management to installation management (see Figure 2.2). Tcl scripts for interfacing with various computing systems are summarized in Table 2.1. An SQL interface to these scripts is described in Table 2.2.

Consider the setup of the WCDMA (wideband code division multiple access) wireless system simulation on `sioux`. The simulation is written in Fortran 95 with a C interface that supports terminal I/O (but not database I/O). The software is parallel and uses MPI for communication. Figure 2.4 illustrates the simulation startup process. We now describe this process in more detail.

The start command (`start_exec_cmd` in `component_installations`) is

```
start_pbs_job -h arabidopsis.cs.vt.edu -U s4w -D s4w -o,
```

where `start_pbs_job` is the name of the Tcl script that submits a batch job to PBS (the batch system used on the cluster), `-h`, `-U`, and `-D` tell the script which database started it, and `-o oid` is the oid of the experiment to run (`oid` is filled in by EMDAG). The `start_pbs_job` script considers two more parameters (`start_exec_arg` column in `component_installations`): `pbs_mpirun wcdma_executor`, where `pbs_mpirun` is the batch script to submit to PBS and `wcdma_executor` is the WCDMA simulation wrapper. Database connection options are passed from one script to another until they are needed in the simulation wrapper. The `pbs_mpirun` script sets up the environment for the `mpirun` command from MPICH (MPI implementation used on the cluster) and passes an appropriate `mpirun` command line to `wcdma_executor`. The latter uses the command line passed by `pbs_mpirun` to start the compiled simulation program called `wcdma`. Then, the wrapper performs database I/O and communicates (via a virtual terminal) with the compiled simulation program.

The corresponding stop command (`stop_exec_cmd` in `component_installations`) is

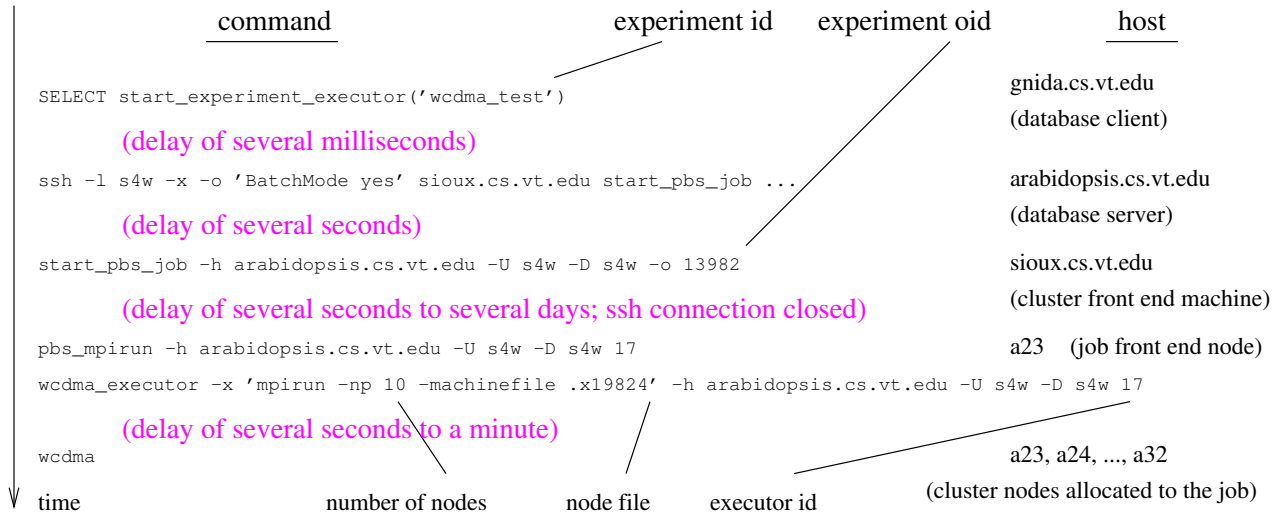


Figure 2.4: WCDMA simulation startup. The WCDMA simulation runs on a Beowulf cluster (sioux.cs.vt.edu) with the database on a machine outside the cluster (arabidopsis.cs.vt.edu) and a database client on yet another machine (gnida.cs.vt.edu). All commands are invoked by EMDAG in response to the client’s SELECT query (first row). The result of this query is the identifier of the started simulation. The query completes as soon as the job is submitted to PBS.

```
stop_pbs_job -h arabidopsis.cs.vt.edu -U s4w -D s4w -n,
```

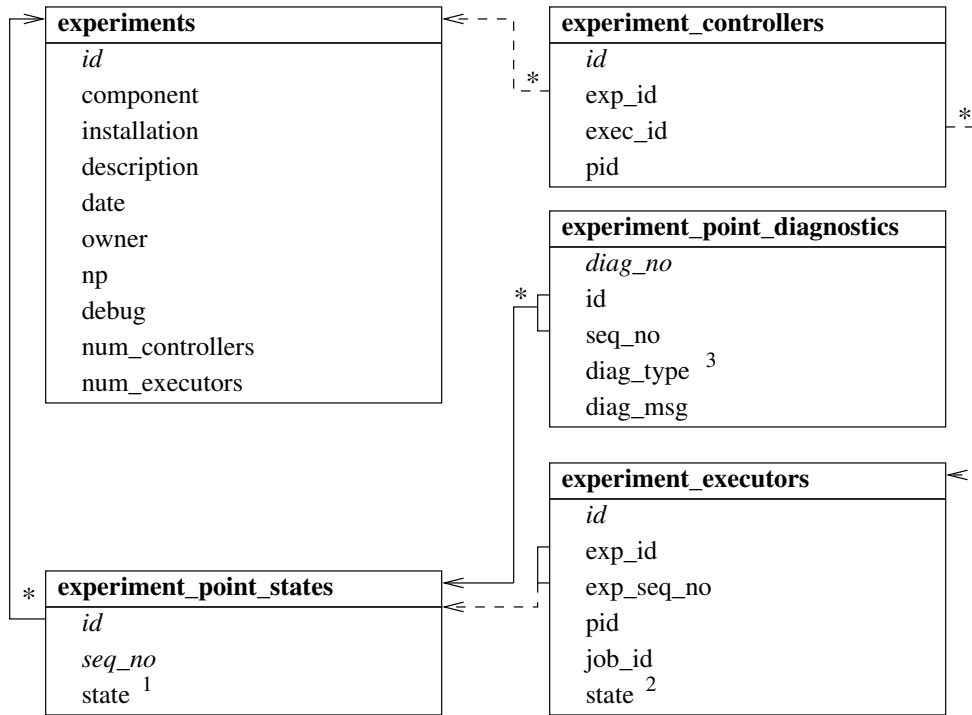
where stop_pbs_job is the script that deletes PBS jobs, -h, -U, and -D are the database connection options (same as above), and -n means that the script must not record executor termination in the database (the caller will do it). EMDAG appends executor id to this command line.

This example shows that while simulation startup is straightforward, it can be tedious to automate. This process is *distributed* because so are the present computing systems. It is quite common to separate the database server from the compute server and to share multiple database and compute servers between a number of research projects. Simulation startup is *asynchronous* because scientific simulations are computationally expensive. Since computing systems are shared by a number of projects, simulation runs are subject to batch scheduling policies. A run may be queued for several days before computing resources are available. Once the computing resources are allocated, the simulation may run for hours to weeks. On the other hand, database queries should not take longer than several seconds. Time-consuming queries tie up database resources and unnecessarily limit data access concurrency. Therefore, simulation startup is distributed and asynchronous.

To summarize, installation management deals with the cumbersome details of interfacing the database to computing systems. Simulation startup and termination are handled by Tcl scripts invoked by the database server process. Large objects are cached in the computing system’s file system for faster access in the future. An SQL interface allows database clients to start and stop simulations on computing systems.

2.1.2 Execution Management

Execution management is the next level of functionality above installation management. Installation management was concerned with interfacing the database to simulation software installed on various computing systems. Execution management concerns with organizing simulation runs, monitoring their status, and



¹ : experiment point states		² : experiment executor states		³ : diagnostic types	
0	fresh	0	queued	0	debug
1	executing	1	idle	1	warning
2	computed	2	running	2	error
3	crashed				

Figure 2.5: Execution management schemas. SQL code for these schemas is available in Appendix A.2.

synchronizing them with each other and with data analysis software. Execution management provides functionality beyond installation management’s interface to computing systems.

The most basic functionality of this kind is simulation status monitoring. The `experiment_executors` table in Figure 2.5 contains up-to-date information on simulation execution status. This table contains one row per running simulation. The `id` column is the simulation run id, `exp_id` and `exp_seq_no` identify the experiment point being computed (or most recently computed), `job_id` is the computing system’s simulation run id (e.g., Unix process id or PBS job id), `state` is the state of the simulation run (queued, waiting for points, or running), and `pid` is described below (this value is used to detect simulation crashes). This table is automatically updated by the simulation startup and termination commands in the previous section. It provides a local view of the status of simulation execution. The next level of execution management functionality is the coordination between several simulation runs.

Parameter sweeps and feedback loops are the two most common (and yet commonly ignored) structures in execution of scientific simulations. Hence, they are prime candidates for abstraction of problem solving in a PSE. Parameter sweeps aim at studying the influence of simulation parameters on the characteristics

function	languages
register_experiment_executor	Tcl, C, SQL
grab_experiment_point	Tcl, C, SQL
experiment_point_diagnostic	Tcl, C, SQL
retrieve_cache	Tcl
store_cache	Tcl
release_experiment_point	Tcl, C, SQL
unregister_experiment_executor	Tcl, C, SQL
register_experiment_controller	SQL
fresh_experiment_point	SQL
unregister_experiment_controller	SQL

Table 2.3: Execution management and installation management functions. These functions are available to executors and controllers in one or more languages: Tcl, C, and SQL. Some Tcl and C functions are implemented on top of the lower level SQL functions.

of the system simulated. One such study is described in a later chapter. Feedback loops can be viewed as a generalization of parameter sweeps. In a feedback loop, the inputs swept over depend on the outputs computed for the previous inputs. Feedback loops arise in applications like numerical optimization, surface fitting, and data mining. Numerical optimization searches the parameter space for an optimal, in some sense, point. Surface fitting applications sample the parameter space to achieve the desired accuracy of the fit. Data mining applications sample parameter spaces in search of patterns. EMDAG's execution management supports two kinds of simulation components—executors and controllers—that implement parameter sweeps and feedback loops.

Recall that the experiment database stores experiments that consist of a number of experiment points (see Figure 2.5). Associated with experiments are experiment executors and experiment controllers. Experiment executors iteratively compute outputs for the experiment points. Pseudo-code for an experiment executor is as follows.

```

while there are fresh points left
  grab a fresh point with the lowest sequence number
  retrieve experiment point inputs from the database
  simulate a system for this set of inputs
  store simulation outputs in the database
  release experiment point
done

```

Likewise, experiment controllers dynamically create new points for the executors to simulate. Pseudo-code for an experiment controller is as follows.

```

while need more points
  create a fresh point with the next available sequence number
  store experiment point inputs in the database
  wait for some executor to compute the outputs for this point
  retrieve the computed outputs and decide what to do next

```

done

Both executors and controllers are software programs implemented by simulation developers. These programs communicate via database tables and database events. A high level interface to the communication facilities is provided via a Tcl package (see Table 2.3). Similar interfaces can be provided for other component topologies. Database tables and events can be used to implement arbitrary communication protocols.

The relationship between executors and controllers in EMDAG can be thought of as a feedback loop or a producer-consumer relationship. There can be many executors and many controllers for any experiment, but there must be no more than one executor per experiment point. Thus, the finest level of granularity in EMDAG is an experiment point. Having multiple executors per experiment is a cheap way to utilize parallel systems. In fact, any simulation that integrates with EMDAG is point-parallel. This feature also allows for a limited form of scheduling. Additional executors can be started for important experiments and executors of unimportant experiments can be stopped. Having multiple controllers per experiment is a cheap way to avoid the overhead of parallel job startup. This feature can be viewed as a poor man's job scheduler. Each executor iteration grabs the fresh point with the lowest sequence number and each controller iteration creates a fresh point with the next available sequence number. Thus, EMDAG's scheduling policy is a fair round-robin. More complicated scheduling policies could be implemented. However, EMDAG was ultimately designed to use external batch systems for scheduling and resource allocation.

In practice, a single software component can be both a controller for some experiment and an executor for another experiment. Controllers are usually parameterized, so it is often desirable to store their parameters and outputs in the database. The performance penalty incurred is easy to justify. The controller should be computationally inexpensive—otherwise, waiting for the executor to compute the point could be a waste of computational resources. However, the corresponding executor must be computationally expensive—otherwise, using EMDAG to manage its data would be a waste of computational resources. Since the controller depends on the executor, the performance penalty for having the database manage the controller's execution and data can be charged to the executor. Since the executor is computationally expensive, this performance penalty is usually acceptable. Therefore, EMDAG does not provide 'start controller' and 'stop controller' functions (see Table 2.2 on page 8). We assume that each controller is also an executor, so controllers can be started by the 'start executor' and 'stop executor' functions.

For example, an optimizer wrapped around a simulation is a controller for the simulation experiment, but an executor for the optimization experiment. The optimization experiment has one point per multiple points in the simulation experiment. Each point in the simulation experiment corresponds to one evaluation of the objective function. Assume that the simulation is computationally expensive. Then, the performance penalty for storing optimizer's data is small compared to the computational expense of multiple objective function evaluations.

Experiment controllers must be started before the corresponding experiment executors. The opposite order of the startup may fail because the executor may terminate before the corresponding controller starts. Executors terminate when (a) there are no fresh points to grab or (b) when no controllers exist for the experiment. By following the above rule one can, in theory, create chains or trees of controller-executor relations. However, this complexity is rarely required in practice.

Limitations of EMDAG

The advantages of EMDAG's approach to execution management are its simplicity, ease of integration with the present computing systems, and ease of deployment on heterogeneous computing architectures. However, this approach has limitations in three areas: transaction management, performance, and error

handling. Some of these limitations are due to the current EMDAG and PostgreSQL implementations. In general, however, data management does not come for free. There is a price to pay for having a database system manage simulation data.

The first limitation is related to transactions. Database transactions cannot be used to protect executors from changes in the state of the database. Doing so will needlessly limit concurrency because simulations are computationally expensive. Lack of ACID semantics (atomicity, consistency, isolation, and durability) places an extra burden on simulation developers. Simulation developers must deal with runtime changes of simulation parameters. Theoretically, this feature can be used to implement computation steering [38]. Practically, this means that an event external to the simulation can cause it to crash. This is a lesser issue with more conventional PSE integration techniques, e.g., plain files or sockets.

Even a process as simple as executor startup exhibits complicated interactions with transaction management. During startup, the database must be accessed by at least three different entities: (i) the client that requested simulation execution, (ii) the script that interfaces with the computing system, and (iii) the executor *per se*. Transaction (ii) is necessary because we must guarantee that the insertion of the executor record is committed before the executor starts. One unpleasant side effect of this transaction is that the executor record may not be visible to transaction (i)—insertion of this record happens in a concurrent transaction. This side effect could be fixed by distributed transaction management (the current version of PostgreSQL does not provide this functionality), but distributed transaction management has problems of its own. We cannot have a single transaction for all three entities because the executor can spend several days in the queue. This delay is too long for a transaction. We cannot get away with two transactions because we must guarantee that the executor will find its record in the database. If transaction *A* spawns a process that starts transaction *B*, there is no easy way to ensure that transaction *A* is committed before transaction *B* starts (short of an intricate play with the database internals). Therefore, EMDAG uses three transactions, commits transaction (ii) before submitting the job to the batch system, and manually rolls it back if job submission fails.

In general, no more than one SQL function that invokes a computing system interface should be called in a single query. Suppose that one function call succeeds but that another call in the same query fails. The database will then be rolled back, but the effect of the first function call on the computing system may not be rolled back. For instance, one can roll back simulation startup, but not simulation termination. Hence, even if PostgreSQL supported exception handling (its current version does not), we would still not be able to guarantee atomicity of some multi-row queries. EMDAG does its best to overcome the impedance mismatch between the database and the computing systems, but this is not always possible.

The second limitation is related to software performance. Compared to traditional PSEs, EMDAG's performance is inferior because simulation inputs and outputs are transferred over the network and written to permanent storage. Likewise, controllers and executors incur synchronization overhead. EMDAG works best for computationally expensive simulations with relatively few outputs. However, performance comparison against traditional PSEs will be misleading. EMDAG provides functionality beyond that found in traditional PSEs. It is well known that extra functionality degrades performance. The real question is how much overhead is due to EMDAG's execution management as opposed to the overhead inherent in database access. EMDAG keeps one record per executor in `experiment_executors` and one record per point in `experiment_point_states`. Any simulation will keep at least one record per point in its inputs table and at least one record per point in its outputs table. Therefore, EMDAG's execution management overhead is, in the worst case, comparable to the inherent overhead of data management. This performance penalty is usually acceptable.

The final limitation of EMDAG's execution management is related to error handling. EMDAG exposes simulation errors that are easy to ignore in conventional PSEs. If an executor crashes, this event will be

recorded in the database, accessible to all users. If a simulation generates incorrect output, this output will be available to everyone in the group. Such situations require human intervention into the otherwise automated execution management system. The cost of such intervention can be relatively high because the user may need to learn EMDAG's internals. Likewise, incorrect simulation output may inadvertently be used by others in aggregation queries. In other words, a shared database provides some of the woes (and joys) of a collaborative PSE. Arguably, the cost of improving software quality must be paid to develop any integrated system [14]. EMDAG is typical in this respect. Human factors aside, let us see the technical limitations of EMDAG's error handling.

First, an executor crash is surprisingly hard to detect. We store executor's database backend id (`pid` column of `experiment_executors`) and periodically check for the existence of the backend process. This check is performed upon any execution activity, e.g., acquisition of a fresh point by some executor. Therefore, EMDAG's tables stay reasonably up to date. (Arguably, they do not need to be up to date if there is no execution activity.) However, this check does not detect all executor crashes. Recall that executor startup is asynchronous (see Figure 2.4 on page 9). Successful job submission to PBS does not imply that the job will ever run. For instance, it can be deleted via an interface external to EMDAG. Even if the job runs, there is no way to detect a failure before the executor connects to the database—executor's backend `pid` is not known until this connection is established. Therefore, there is a class of executor crashes that EMDAG cannot even detect. It is the user's responsibility to update EMDAG's tables in these cases. Users can inspect the log files of the job startup scripts (see Table 2.1 on page 8) to debug such errors.

Second, even if an executor crash is detected, EMDAG cannot take any corrective action. The crash can be due to a transient condition (e.g., a network failure or an unusual sequence of random numbers) or due to a permanent condition (e.g., a misconfiguration or a programming error). The point should be recomputed if the error condition is transient. However, doing so when the error condition is permanent is not a wise course of action. EMDAG has a special execution state for crashed points. Crashed points are ignored by experiment executors. An appropriate action must be taken and point states must be reset by the user. The SQL function `reset_crashed_points()` resets the state of all crashed points in a given experiment to 'fresh' and erases all diagnostic messages for these points. If the simulation computed incorrect results, `reset_all_points()` can be used to reset both computed and crashed points. These functions only reset the execution state of the points. It is the user's responsibility to erase the partial simulation outputs (if desired).

To summarize, EMDAG's execution management is designed to support parameter sweeps and feedback loops in a natural way. Each simulation component in EMDAG is either an experiment executor or an experiment controller (or both). Executors and controllers communicate via a standard protocol built on top of database tables and events. EMDAG automates the most commonly needed (in our experience) execution management functionality in a PSE. Some development effort is necessary to integrate an existing simulation with EMDAG. However, most of this effort is inherent in designing any data management system for the simulation.

2.1.3 Data Management

Unlike ZOO, EMDAG does not provide data management abstractions beyond those available in the underlying relational database. Therefore, our generic discussion of data management will be brief. Ultimately, schema design for a particular simulation is the responsibility of the simulation developers. This subsection describes several schema design conventions that should be followed for integration with EMDAG.

According to Figure 2.6, EMDAG only requires that all simulation data be organized as experiments and that all data within an experiment be organized as points. Table names `experiment_inputs` and `experiment_outputs` are place holders for component-specific tables. Even this mild assumption

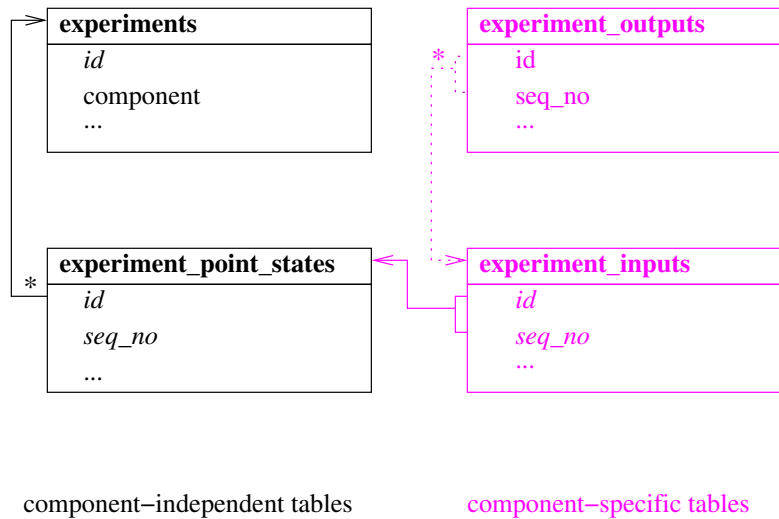


Figure 2.6: Data management schemas. The following section provides examples of component-specific schemas in S⁴W.

can be violated. E.g., some simulations may need bootstrap tables that are unrelated to EMDAG’s tables. EMDAG never queries simulation data, so simulation schemas are largely irrelevant to EMDAG.

However, thinking of simulation data as point inputs and outputs helps one understand the dynamics of the EMDAG system. The inputs to a parameter sweep are prepared and stored in the database during the experiment design, i.e., before the simulation is executed. The outputs are then computed by experiment executors point by point. Therefore, at any given moment, the database contains a subset of the outputs for the stored inputs. This is different from the approach taken by ZOO where only some special status information is available for the in-progress experiments. EMDAG was designed with simulations in mind. It has been our experience that parameter sweeps are rarely executed till completion. The simulation is usually stopped once the computed results exceed the reasonable range. Therefore, enforcing database consistency at every point during simulation execution is important to ensuring system usability. Few simulations run to completion, so the ‘final’ results are often partial.

Another difference of EMDAG from ZOO is a more judicious use of triggers in EMDAG. In particular, EMDAG does not interpret insertion of simulation inputs into a table as a command to start an executor. First, it may be advantageous to decouple experiment definition from experiment execution. E.g., a user may wish to save some partial inputs and pause to look up a few constants or ask for help. Likewise, a failure in experiment execution should not cause a failure in storage of experiment inputs. Users should be able to store experiment inputs when the parallel computing system is down. Second, it is unwise to submit parallel jobs without explicit user consent. When the computing resources are shared, users should have full control over the number and kinds of executors they start. Finally, correct implementation of such triggers is technically difficult. E.g., an experiment executor started by a trigger may execute before all inserts of simulation inputs are committed. Triggers also complicate the database dump and restore operations.

EMDAG uses triggers for minor administrative tasks. One such task is garbage collection. E.g., all simulation data references (directly or indirectly) the `experiment_point_states` table and the latter references the `experiments` table via foreign keys. When a record is deleted from `experiments`, the foreign key trigger removes the corresponding records from all other tables. Thus, deleting a record from `experiments` deletes the entire experiment. Triggers are also used to implement custom access control.

E.g., a trigger on `experiments` does not allow anyone but the experiment owner or the super user to delete an experiment. Likewise, only the super user can change the experiment owner. Finally, triggers are used to provide event notification. E.g., a trigger on `experiment_point_states` fires appropriate events on updates to the `state` column.

This subsection described several data management conventions in EMDAG. EMDAG tries to be unobtrusive in this respect. The simulation developers are free to define their own schemas. We also outlined important differences of EMDAG from ZOO. EMDAG's functionality is specialized for simulation data management, so we think that EMDAG is more useful than ZOO in this setting. The following section describes data management for S⁴W components, an example application of EMDAG.

2.2 Computation and Data Models in S⁴W

This section describes the simulations and the data models currently available in S⁴W. The material is organized into three subsections: the ray tracing propagation model, the WCDMA wireless system model, and the DIRECT numerical optimizer. Each subsection begins with a general description of the computational component. Then, we present the schemas for the component's data and justify our data modeling choices. Our emphasis is on the integration of realistic simulation components with our experiment management system (EMDAG) and a relational database (PostgreSQL).

2.2.1 Ray Tracing Model and Data Management

Ray tracing is a popular high-fidelity deterministic model of wireless signal propagation. This model estimates average signal powers and channel impulse responses at receiver locations of interest. The popularity of ray tracing is due to its wide applicability and relatively high prediction accuracy. This subsection outlines the ray tracing propagation model and data management in S⁴WRT, a parallel 3D ray tracer for S⁴W.

Ray tracing has been extensively studied in the literature [47]. This subsection does not cover environment modeling, computational geometry, or load balancing. We emphasize (a) the conversion of S⁴WRT predictions to match the characteristics of a real wireless system and (b) the data models that enable the use of ray tracing in scientific experiments. In other words, this subsection deals with the integration issues.

Ray Tracing Model

S⁴WRT models electromagnetic waves as pyramidal beams. The beams are shot from geodesic domes drawn around transmitter locations. Each beam is a triangular pyramid formed by the point location of the transmitter and a triangle on the surface of the dome. Beams are traced through reflections and penetrations through the walls in a particular environment. Once an intersection with a receiver location is detected, a ray is traced back from the receiver location to the transmitter location through the sequence of reflections and penetrations encountered by the beam. The illustration of this process in 2D is given in Figure 2.7. Figure 2.8 shows an intersection test of a beam with a grid of uniformly spaced receiver locations. S⁴WRT does not currently include either diffraction or scattering. These phenomena play an important role in propagation, but their simulation is computationally expensive. Octree space partitioning and image parallelism with dynamic scheduling [25] are used to further reduce simulation run time.

Suppose that a receiver location is hit by N rays $1, 2, \dots, N$. Each ray is characterized by an arrival time t_j , a power P_j , a phase ϕ_j , transmission angles $(\varphi_j^{(t)}, \theta_j^{(t)})$, and reception angles $(\varphi_j^{(r)}, \theta_j^{(r)})$. The time t_j of the j -th ray, in seconds, is calculated from the total distance d_j traveled by the ray, in meters, and

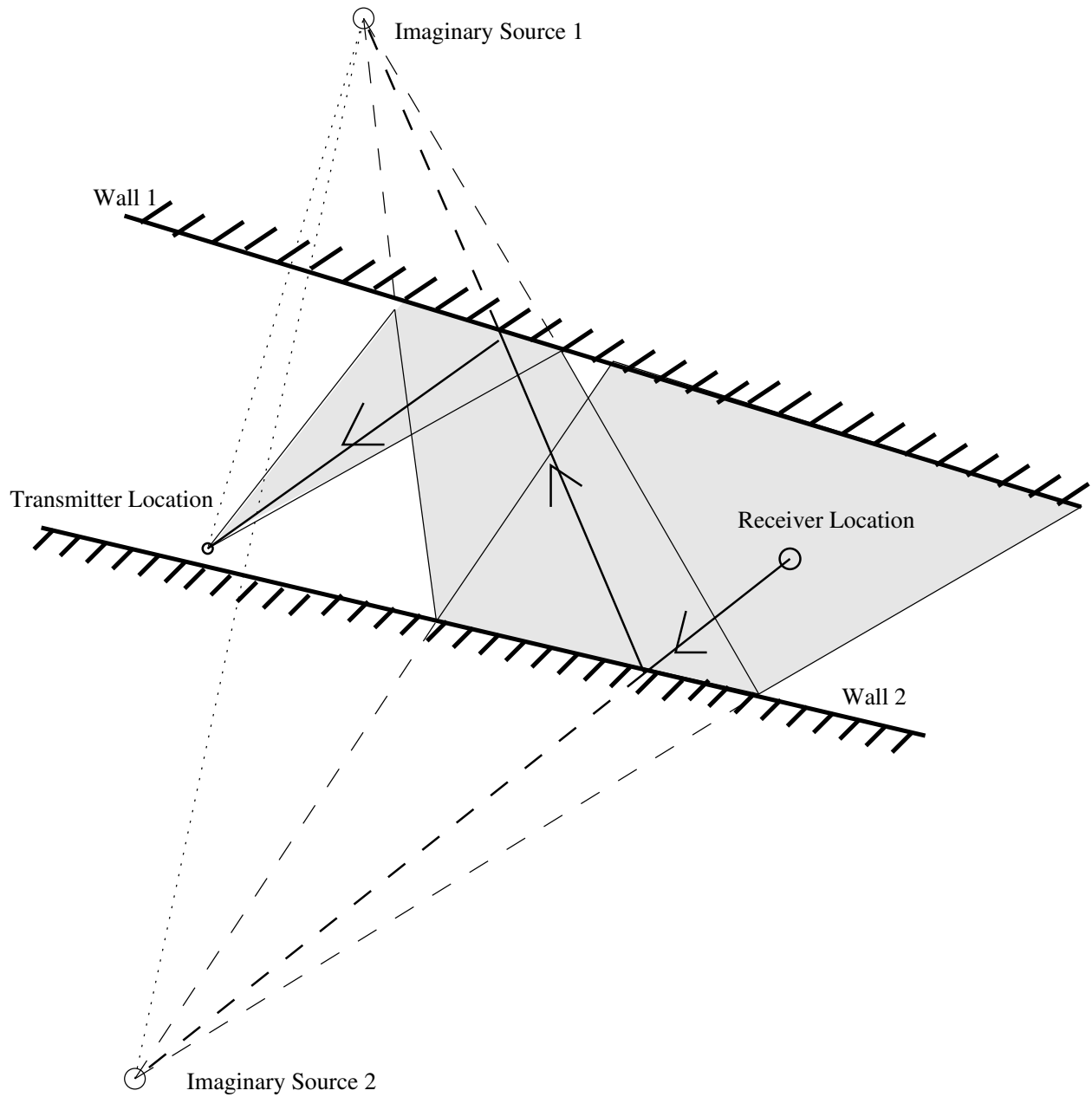


Figure 2.7: 2D beam tracing. A beam (shaded region) is traced from the transmitter location to the receiver location through two reflections. Then, a ray (bold line) is traced back.

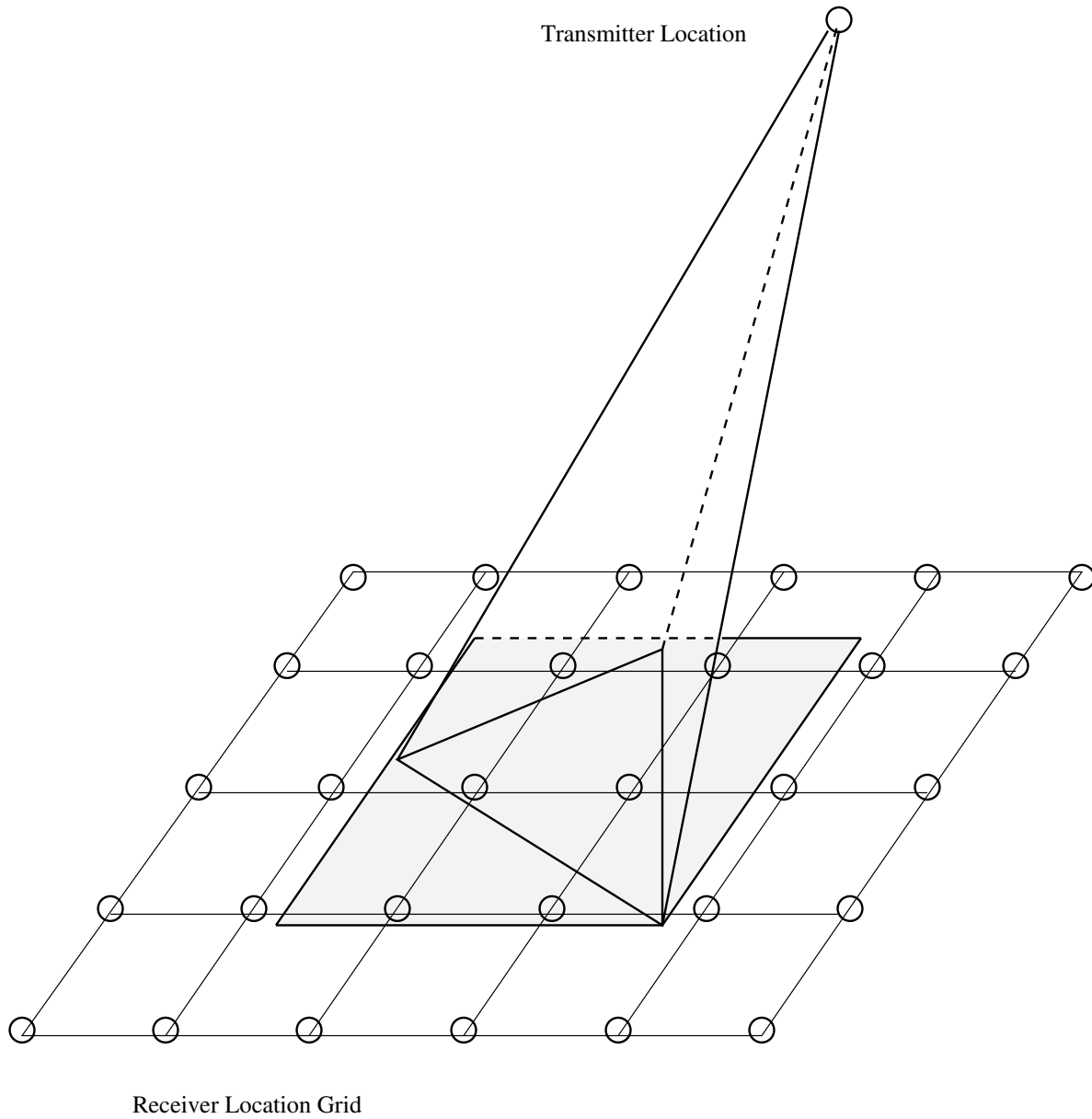


Figure 2.8: Beam intersection with a receiver location grid. Only the locations inside the bounding box of the projection of the beam onto the grid (shaded region) are tested for intersection with the beam pyramid.

the speed of light $c = 299792458$ meters per second, as

$$t_j = d_j/c.$$

The power contribution P_j of the j -th ray, in dBW, is calculated according to the model developed in [47] as

$$P_j = P(d_0) - 20 \log_{10}(d_j/(d_0\lambda)) - nL_r - mL_t,$$

where $P(d_0)$ is the transmitter power at a reference distance d_0 from the transmitter ($d_0 = 1$ m for indoor applications), n and m are the numbers of reflections and wall penetrations encountered by the ray, L_r and L_t are constant reflection and penetration losses, in dB, and λ is carrier wavelength, in m. The phase ϕ_j of the j -th ray is calculated as

$$\phi_j = n\pi + 2\pi d_j/\lambda.$$

Finally, the angles $(\varphi_j^{(t)}, \theta_j^{(t)})$ and $(\varphi_j^{(r)}, \theta_j^{(r)})$ are calculated from the floor plan geometry. This simple model assumes constant reflection and penetration losses and a 180 degree phase shift per reflection.

The impulse response predicted by this model can be written as

$$h(t) = \sum_{j=1}^N E_j e^{i\phi_j} \delta(t - t_j),$$

where $E_j = \sqrt{\eta 10^{0.1P_j}/A_e}$ is the magnitude of the received E -field envelope of ray j , in V/m, $\eta = 120\pi \Omega$ is the impedance of free space, A_e is the effective aperture of the antenna, in m^2 , and $\delta(\cdot)$ is the ‘ δ -function’ (the latter is a common abuse of notation). Assume that the reference power $P(d_0)$ was measured in the direction of both the maximum transmitter antenna gain and the maximum receiver antenna gain and that the maximum antenna gains of both devices were 0 dBi. Then, $h(t)$ is the impulse response of a hypothetical measurement system with isotropic antennas of unity gain and an infinite bandwidth resolution.

Application of antenna patterns is the first step to making the impulse response $h(t)$ match the output of a real measurement system. Application of antenna patterns changes the magnitude E_j of the received E -field envelope to

$$E'_j = E_j G_t(\varphi_j^{(t)}, \theta_j^{(t)}) G_r(\varphi_j^{(r)}, \theta_j^{(r)}),$$

where $G_t(\varphi_j^{(t)}, \theta_j^{(t)})$ and $G_r(\varphi_j^{(r)}, \theta_j^{(r)})$ are the transmitter and the receiver antenna gains in the directions of ray departure and arrival, respectively. S⁴WRT can calculate the gains for omnidirectional, waveguide, pyramidal horn, and biconical antennas. The gain formulas for the first three antenna types can be found in [48]. The patterns for the waveguide and the pyramidal horn are interpolated to 3D from 2D principal plane patterns. The biconical antenna pattern is interpolated to 3D from a set of measurements. The resultant impulse response

$$h'(t) = \sum_{j=1}^N E'_j e^{i\phi_j} \delta(t - t_j)$$

corresponds to a hypothetical measurement system with real antennas but still an infinite bandwidth resolution.

To account for a finite bandwidth resolution, we transmit a Gaussian pulse $g(t)$ through the channel with the impulse response $h'(t)$. The same result can be obtained by replacing the infinitely thin unit pulse $\delta(t)$ with the Gaussian pulse $g(t)$ of finite *pulse width* 3σ . Let the Gaussian pulse be

$$g(t) = e^{-t^2/(2\sigma^2)},$$

where t and σ are in seconds. The response $y(t)$ of the linear time-invariant channel, with the impulse response $h'(t)$, to the Gaussian pulse $g(t)$ is the convolution of $h'(t)$ and $g(t)$, namely,

$$\begin{aligned} y(t) &= \int_{-\infty}^{\infty} h'(\tau)g(t-\tau)d\tau \\ &= \int_{-\infty}^{\infty} \sum_{j=1}^N E'_j e^{i\phi_j} \delta(\tau-t_j) e^{-(t-\tau)^2/(2\sigma^2)} d\tau \\ &= \sum_{j=1}^N E'_j e^{i\phi_j} e^{-(t-t_j)^2/(2\sigma^2)}, \end{aligned}$$

where the last transformation is due to the well known sifting property of the ‘ δ -function’.

Finally, we sample the impulse response $y(t)$ at uniform time intervals of width Δ , where the *bin width* Δ is the bandwidth resolution of the measurement system, in seconds. Assume that the measurement system triggers on the first peak and that the line-of-sight (LOS) signal is strong enough to be detected. Then, we can align the center of the first time bin with the arrival time of the first ray. This time t_0 is referred to as the *reference time*. The measured electric field $E_k^{(m)}$ at time $t_0 + k\Delta$ is

$$E_k^{(m)} = y(t_0 + k\Delta) = \sum_{j=1}^N E'_j e^{-(t_0+k\Delta-t_j)^2/(2\sigma^2)} e^{i\phi_j}.$$

The power in time bin k centered at time $t_0 + k\Delta$ is approximated as

$$P_k^{(m)} = 10 \log_{10}(|E_k^{(m)}|^2 A_e / \eta),$$

where $|\cdot|$ denotes the magnitude of the complex number and the constants η and A_e are defined above. Strictly speaking, this expression should be integrated over the duration of the time bin. The point approximation above is similar to the one employed in the measurement system used for S⁴WRT calibration [8].

The constants in these equations deserve special attention. Since all impulse response transformations are linear, the constants A_e and η that relate the power and the electric field do not affect the final result. These constants are ‘stripped off’ of P_j at the beginning and re-introduced into $P_k^{(m)}$ at the end. Most applications are not concerned with the E -fields, so knowledge of the effective antenna aperture A_e is rarely required. An astute reader might have noticed that the Gaussian pulse $g(t)$ is normalized with respect to the peak $g(0)$, not with respect to the area $\int_{-\infty}^{\infty} g(t)dt$ or the squared area $\int_{-\infty}^{\infty} g^2(t)dt$. One can use a narrowband result ($N = 1$ ray) to verify the correctness of this normalization. Assuming unity antenna gains, the narrowband power P_1 must be equal to the final outcome $P_0^{(m)}$ of the transformations described here.

Ray Tracing Data Management

Two sets of schemas can be identified for S⁴WRT. The first set of schemas (see Figure 2.9 on page 22) deals with environment preprocessing. Preprocessing consists of (a) converting floor plans from Autocad’s DXF data format to the XML format understood by S⁴W, (b) triangulating the polygons (walls) in the floor plan, and (c) partitioning the triangles into voxels to improve the speed of ray tracing. The second set of schemas (see Figure 2.10 on page 23) deals with ray tracing *per se*. These schemas describe device locations, device parameters, and propagation simulation parameters. Each run of S⁴WRT simulates impulse

responses for a number of channels in a given environment. S⁴WRT supports multiple transmitter and receiver locations each equipped with multiple antennas. This software has been used in studies of space-time transmit diversity [10] and optimal transmitter placement [50].

Figure 2.9 describes the three preprocessing steps. The tables on the left of this figure contain the inputs to the preprocessing software. The tables on the right of this figure contain the outputs of the preprocessing software. The preprocessing components are chained top to bottom. The bootstrap table `dxs_environments` contains the floor plans in their most primitive form. The Tcl script `store_dxs` stores Autocad's DXF files as compressed PostgreSQL large objects and inserts appropriate records into `dxs_environments`. Then, users insert records into the input tables on the left, cross-reference these records to the output tables one level above, and invoke the preprocessing software (EMDAG's executors). The executors perform preprocessing, store the resultant floor plans as PostgreSQL large objects, and insert appropriate records into the output tables on the right. The output tables contain references to the preprocessed environments and summary information. Thus, the preprocessing tables are filled in top-to-bottom and left-to-right.

Triggers automate common database access tasks. E.g., inserting a record into any input table will insert a 'fresh point' record into the `experiment_point_states` table (if it does not already exist). Likewise, deleting a record from any table will delete all records that (directly or indirectly) reference the deleted record. Also, deleting a record from any output table will delete the corresponding large object (floor plan) from the database. Therefore, any preprocessing run can be deleted by deleting its top-level record (e.g., the record in `experiments`). The database will then delete all records that are part of, result of, or depend on the deleted run.

The only non-trivial preprocessing component is `boct`. This component performs octree partitioning [29] of the environment. Space partitioning is heuristic. Users must adjust `boct`'s parameters (`min_dx`, `max_triangles`, `max_dup`, and `c_scale`) to achieve a good quality partitioning of the environment. Once a good partitioning of the environment has been computed, it can be reused in all subsequent ray tracing simulations. Users sweep over the values of `boct`'s parameters and then choose the octree with good summary information. (Our experiments indicate that the ray tracing software performance is maximal when the octree contains an average of ten triangles per voxel.) Octrees are large and fast to recompute, so they are not stored in the database. Instead, the `boct_inputs` table contains the necessary parameters and the `octree_environments` table contains the octree statistics. The ray tracer reads and broadcasts the triangular environment. Then, each processor recomputes the octree using the parameters in `boct_inputs`. The most recently used octree is cached between the ray tracing points (runs). The broadcasting and the partitioning are repeated only when the octree identification changes.

Preprocessing components are serial programs that run on Unix workstations. All of these programs are written in C and provide command-line interfaces. Wrapper scripts, written in Tcl (Expect), handle the database I/O and the integration with EMDAG's execution management. The ray tracer is a parallel program written in C. It uses MPI for inter-node communication, the C interface to PostgreSQL for database I/O, and the C interface to EMDAG for integration with EMDAG's execution management.

The ray tracing schemas are shown in Figure 2.10. The inputs to each ray tracing run (EMDAG's point) are (a) the octree, (b) a number of transmitter locations, (c) a number of receiver locations, and (d) a number of grids of receiver locations. Each transmitter or receiver location can be equipped with multiple antennas. S⁴WRT records an impulse response (power delay profile) per pair of transmitter location and receiver location antennas.

The `antennas`, `waveguide_antennas`, and `horn_antennas` tables contain antenna pattern parameters. The octree tables and the antenna pattern tables are the bootstrap tables for the ray tracer. The records in these tables are not strictly part of the ray tracing simulations. However, these records must exist

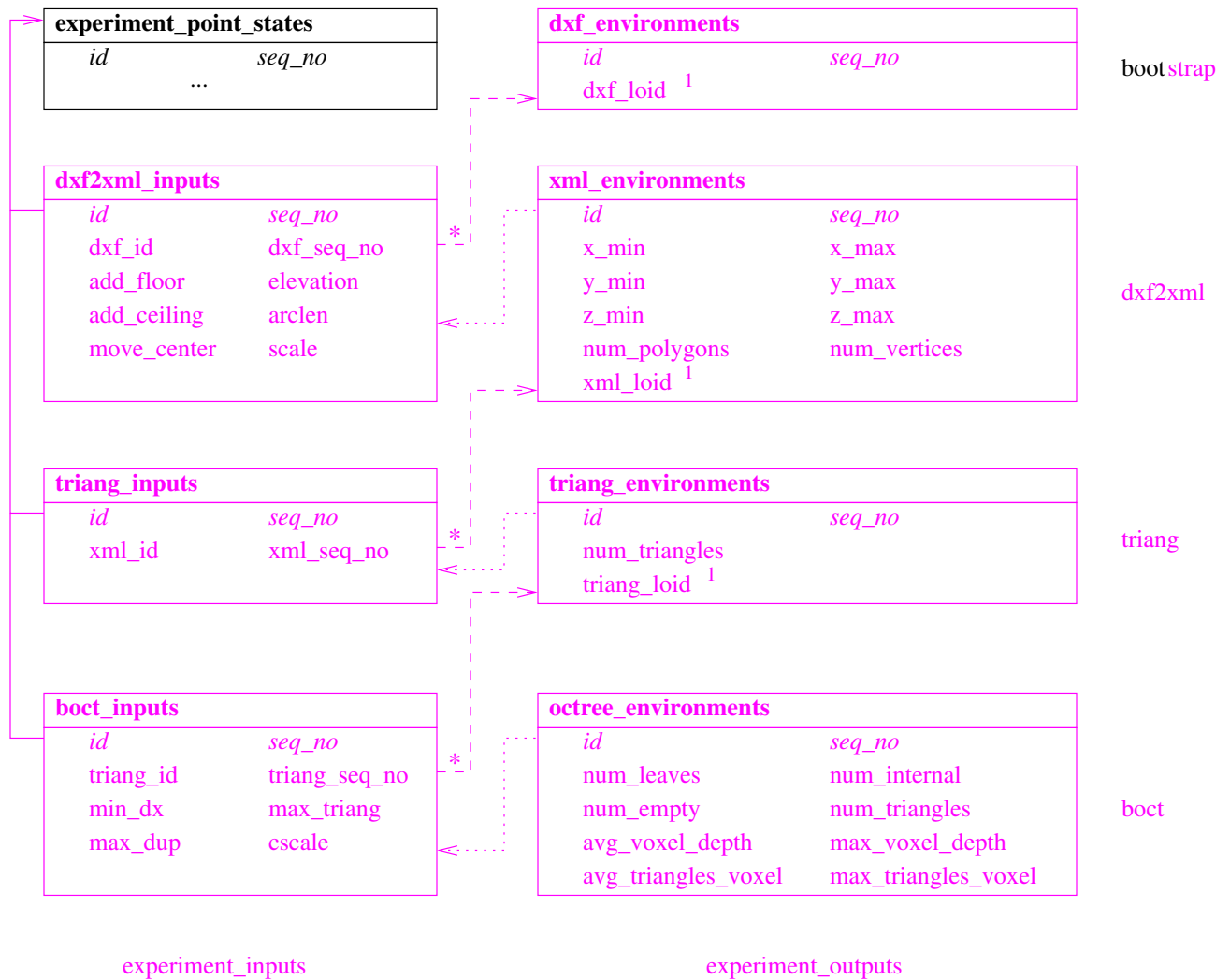


Figure 2.9: Environment preprocessing schemas. These tables are filled in top-to-bottom and left-to-right. SQL code for these schemas is available in Appendix B.1.

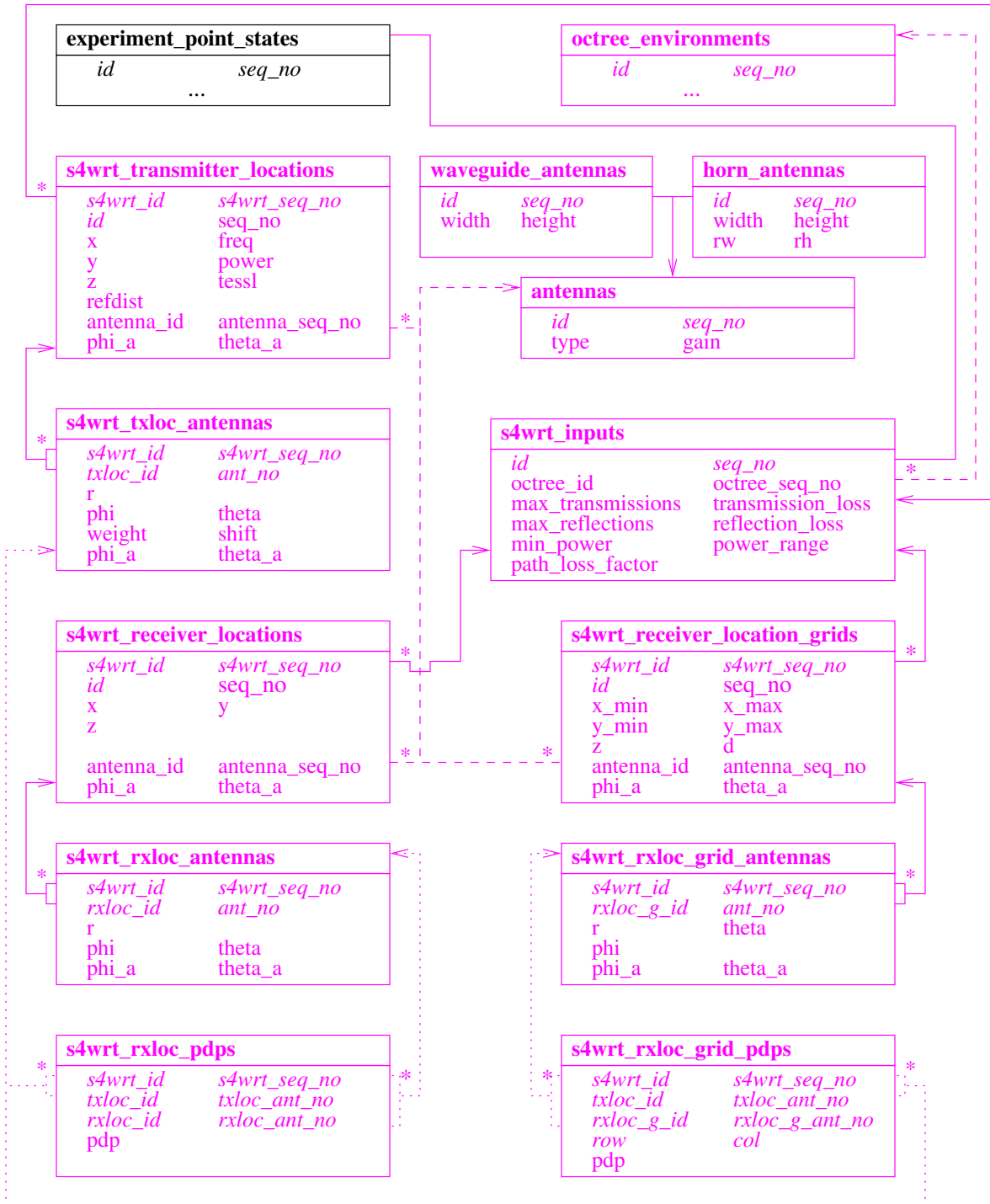


Figure 2.10: Ray tracing schemas. SQL code for these schemas is available in Appendix B.1.

function	description
<code>reference_time(raw_pdp)</code>	arrival time of the first ray, in ns
<code>num_rays(raw_pdp)</code>	the number of rays
<code>width(raw_pdp)</code>	the time span between the first and the last rays, in ns
<code>peak_power(raw_pdp)</code>	the maximum of ray powers, in watts
<code>total_power(raw_pdp)</code>	the sum of ray powers, in watts
<code>mean_excess_delay(raw_pdp)</code>	see [45]
<code>rms_delay_spread(raw_pdp)</code>	see [45]

Table 2.4: SQL functions for ‘raw’ impulse responses.

for the ray tracing to proceed. The antenna pattern tables contain antenna types, antenna parameters, and maximum gains. The maximum gains are in dB w.r.t. the power (recall that the gains $\{G_{t,r}(\varphi_j^{(t),(r)}, \theta_j^{(t),(r)})\}$ were in linear units w.r.t. the E -field). The `type` column of antennas can take on the following values: 0 (isotropic), 1 (omnidirectional), 2 (biconical), 3 (waveguide), or 4 (pyramidal horn). The latter two antenna types require antenna dimensions. These dimensions are stored in the `waveguide_antennas` and `horn_antennas` tables. Antenna dimensions are in millimeters.

The `s4wrt_transmitter_locations` table contains one record per transmitter location. This record contains the reference values for all transmitter location antennas: reference coordinates (x,y,z) , reference frequency `freq`, in MHz, total power `power`, in dBW, and reference antenna orientation in polar coordinates $(\text{phi}_a, \text{theta}_a)$, in radians. The `s4wrt_txloc_antennas` table contains the per-antenna offsets from these reference values: the location offset $(r, \text{phi}, \text{theta})$, the frequency offset `shift`, the power weight `weight`, and the antenna orientation offset $(\text{phi}_a, \text{theta}_a)$. The power is allocated to antennas according to their linear weights. The sum of all antenna powers, in watts, equals the total transmitter location power. The total transmitter location power is the reference power $P(d_0)$ less the effect of the isotropic antennas, i.e.,

$$\text{power} = P(d_0) + 20 \log_{10}(4\pi) \approx P(d_0) + 22,$$

where $P(d_0)$ is in dBW. All other offsets are added to their reference values. S⁴WRT assumes that all antennas at the same transmitter location have the same antenna pattern (`antenna_id, antenna_seq_no`), the same reference distance `refdist` (d_0), in m, and the same geodesic tessellation frequency `tessl` ($20 \times (\text{tessl})^2$ beams are traced per antenna).

The `s4wrt_receiver_locations` and `s4wrt_rxloc_antennas` tables contain similar information for receiver locations. In addition to individual receiver locations, S⁴WRT supports grids of receiver locations with uniform elevation `z` and X and Y separation `d`, in meters. All receiver locations on the grid have identical parameters, except, obviously, for their spatial coordinates. Grids of receiver locations are used in coverage studies.

S⁴WRT records the ‘raw’ impulse responses $\{h'(t)\}$. One impulse response is recorded per pair of transmitter location and receiver location antennas. An impulse response consists of a number of rays. Recall that each ray is characterized by an arrival time, a power, a phase, arrival angles, and departure angles. An impulse response is an indivisible unit of data in propagation modeling. Therefore, custom PostgreSQL data types have been implemented for impulse responses. S⁴WRT output tables use the `raw_pdp` data type, which is a list of seven-tuples of ray attributes. The time is in nanoseconds, the power is in watts, the phase is in radians relative to the transmitter location, and all ray angles are in radians. The rays are sorted in ascending order of their arrival times. ‘Raw’ impulse responses predict the output of a hypothetical

measurement system with an infinite bandwidth resolution. Table 2.4 lists the SQL functions that compute various statistics of ‘raw’ impulse responses.

The `raw_pdp` data type represents a compromise between several contradictory objectives. On the one hand, we should save as little information as possible in the database. Saving detailed outputs degrades software performance, especially for closely spaced grids of receiver locations. On the other hand, we should save sufficient information for subsequent data analysis. E.g., saving the peak powers alone is sufficient for the power coverage optimization, but insufficient for calibration and for the WCDMA simulation. A related issue is the flexibility of the data format. E.g., resampling the impulse responses at a given bandwidth resolution Δ usually reduces storage requirements, but precludes us from studying the effects of changing the bandwidth resolution of the system (resampling is a one-way transformation). The right compromise between the generality and the efficiency of the data format ultimately depends on the uses of the software.

S⁴WRT has been developed and used in a research environment. During the two-year life time of this software, the author’s view of the simulation, the uses of the simulation by other project participants, and even the research goals of the project have changed an enumerable number of times. Therefore, the current schemas for the S⁴WRT data by far and large sacrifice efficiency for the sake of generality. Ultimately, the amortized incremental cost of storing more data than necessary is significantly smaller than the cost of developing custom data formats for particular applications. S⁴WRT stores all information it computes except for ray paths. It does apply antenna patterns to the impulse responses, but the stored impulse responses contain sufficient information to undo this transformation. Such detail in impulse response modeling is unnecessary for any given application, but is required for the union of all functionalities expected of a ray tracing propagation model. The S⁴WRT data model in Figure 2.10 serves as an ‘orthogonal basis’ for the functionalities presented next.

2.2.2 WCDMA Simulation Model and Data Management

The ray tracing propagation model predicts the impulse response of a wireless channel. However, propagation models do not directly predict the performance of any particular wireless system that operates in this channel. S⁴W provides a Monte Carlo simulation of WCDMA (wideband code division multiple access) wireless systems. The WCDMA simulation predicts the BER (bit error rate) of a WCDMA system that operates in a channel with a given impulse response. This section briefly outlines the WCDMA simulation and describes its data format. We also explain the use of ray tracing results in WCDMA simulations.

WCDMA Simulation

The WCDMA simulation predicts the BER of a wireless system that operates in a given channel. Figure 2.11 briefly describes the computational steps of the WCDMA simulation. The random information source part of the transmitter module generates random information bits to be sent through a wireless channel. The generated information is processed with a series of digital signal processing techniques to reduce the number of potential errors. The wireless channel is modeled as a linear Rayleigh fading filter with a given impulse response. Before being sent to the receiver, the channel output is combined with the Gaussian noise from the electronic system. The received distorted signal is processed with a series of digital signal processing techniques by the receiver module. Finally, the estimated information bits are compared with the original information bits for the BER. [50]

The WCDMA simulation is implemented in Fortran 95. It uses MPI for inter-node communication. A C interface to this simulation provides terminal I/O. Database I/O and integration with EMDAG are provided by a Tcl (Expect) wrapper script.

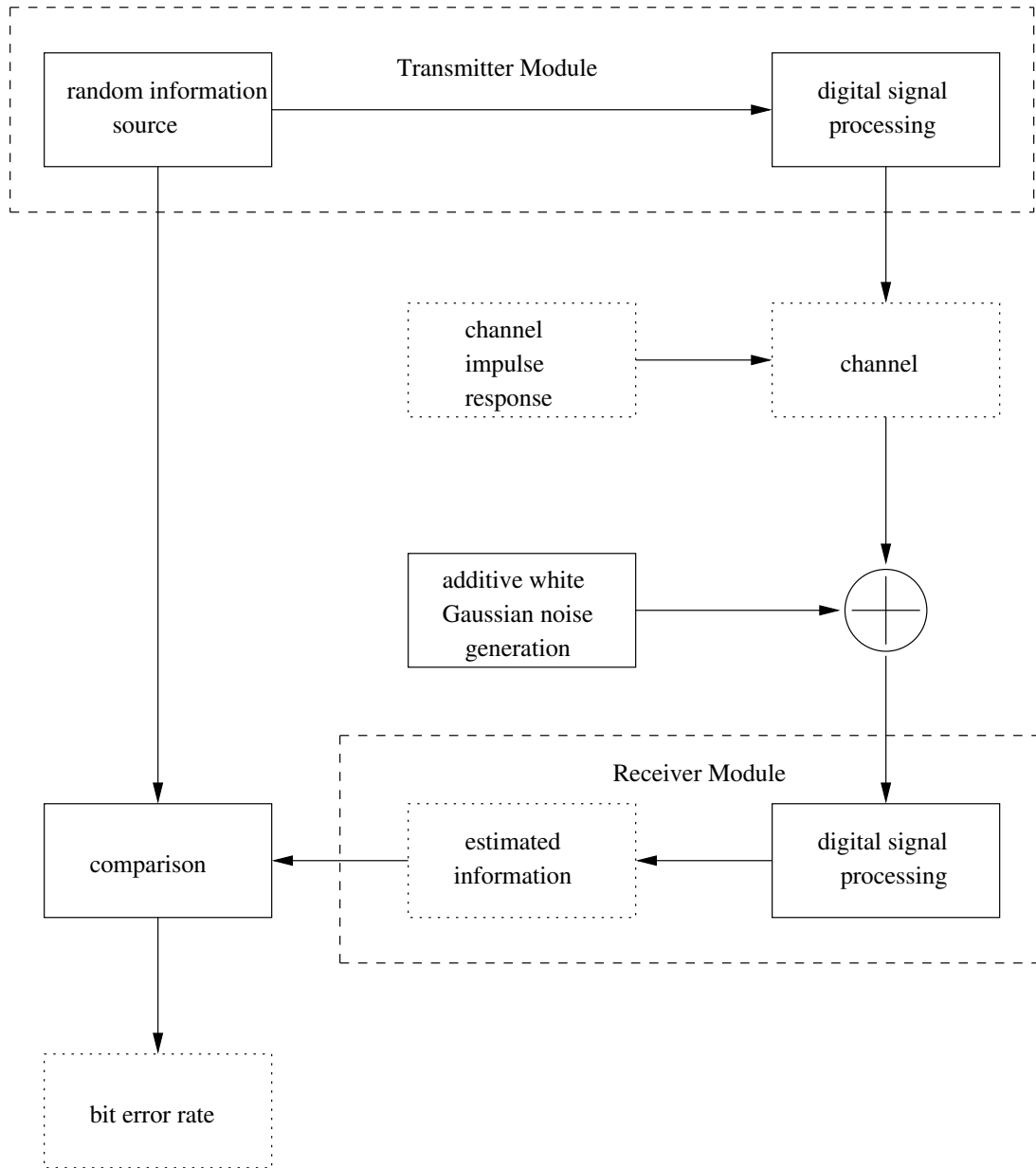


Figure 2.11: Block diagram of the WCDMA simulation. The solid boxes are the software modules and the dotted boxes are the data.

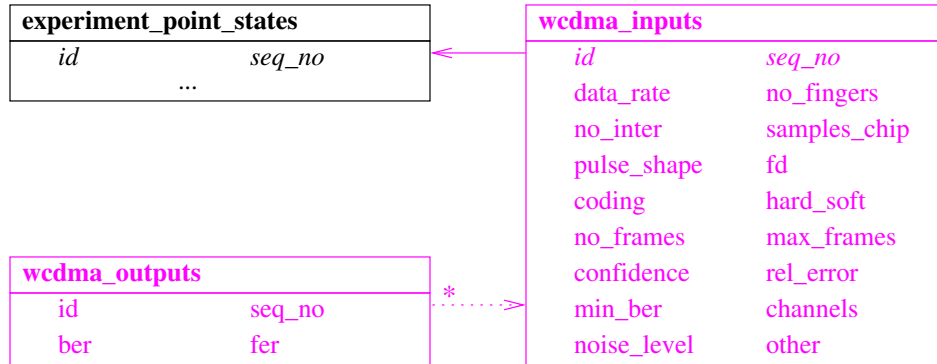


Figure 2.12: WCDMA simulation schemas. SQL code for these schemas is available in Appendix B.2.

function	description
<code>resample(raw_pdp, bin_width, pulse_width)</code>	see subsection 2.2.1
<code>resample(raw_pdp, bin_width)</code>	same, <code>pulse_width = bin_width</code>
<code>reference_time(sampled_pdp)</code>	arrival time of the first bin, in ns
<code>bin_width(sampled_pdp)</code>	bandwidth resolution, in ns
<code>num_bins(sampled_pdp)</code>	the number of bins with non-zero power
<code>width(sampled_pdp)</code>	one more than the index of the last bin
<code>nth_power(sampled_pdp, n)</code>	power of bin <i>n</i> , in watts
<code>peak_power(sampled_pdp)</code>	the maximum of bin powers, in watts
<code>total_power(sampled_pdp)</code>	the sum of bin powers, in watts
<code>mean_excess_delay(sampled_pdp)</code>	see [45]
<code>rms_delay_spread(sampled_pdp)</code>	see [45]
<code>trim_bins(sampled_pdp, threshold)</code>	remove bins with relatively small power
<code>resample(sampled_pdp, bin_width)</code>	heuristic resampling

Table 2.5: SQL functions for ‘sampled’ impulse responses.

WCDMA Simulation Schemas

Figure 2.12 depicts the schemas for the WCDMA simulation data. The `wcdma_inputs` table contains the inputs to the WCDMA simulation. The `wcdma_outputs` table contains the simulation outputs.

Most of the WCDMA simulation inputs are hardware parameters: the data rate, the number of fingers in the RAKE receiver, etc. A notable exception is the `channels` column. This column contains an array of impulse responses of the transmitter antennas at the receiver location of interest. The simulation supports up to two transmitter antennas. The impulse responses are of PostgreSQL data type `sampled_pdp`. This data type is used for impulse responses measured by the wireless system being simulated, i.e., a system with finite bandwidth resolution Δ (see subsection 2.2.1). Utility SQL functions for `sampled_pdp` can be found in Table 2.5. Two of these functions calculate a ‘sampled’ impulse response from a ‘raw’ impulse response output by the ray tracer.

The impulse responses predicted by ray tracing do not contain the effects of thermal noise. However, noise energy is required to obtain the signal-to-noise ratio (SNR) for the WCDMA simulation. The `noise_level` column of `wcdma_inputs` contains the noise at the output of the receiver, in watts. Ac-

ording to [45], the noise level can be calculated as

$$N_0 = GFkT_0B,$$

where G is the overall receiver gain (in linear units), F is the noise figure of the receiver (in linear units), k is the Boltzmann's constant (in Joules per Kelvin), T_0 is the room temperature (in Kelvins), and B is the effective bandwidth of the receiver (in Hertz). E.g., a typical 802.11b wireless card has a gain $G = 30$ dB and a noise figure $F = 5$ dB. The bandwidth of an 802.11b channel is usually $B = 22$ MHz. The room temperature can be taken as $T_0 = 300$ K (75°F, 27°C). Then, the noise level N_0 is

$$N_0 = (10^{3.0}) \times (10^{0.5}) \times (1.38 \times 10^{-23}) \times (300) \times (22 \times 10^6) \approx 2.88 \times 10^{-10} \text{ watts} \approx -95.4 \text{ dBW}.$$

Finally, the SNR S for the WCDMA simulation, in dB, is defined as

$$S = \left(\max_{0 \leq k < M} P_k^{(m)} \right) - N_0,$$

where M is the number of bins in the impulse response, $P_k^{(m)}$ is the power of the k -th bin, in dBW, and N_0 is the noise level, in dBW.

The WCDMA simulation is computationally expensive. Practical values for the BER are 10^{-3} for voice quality applications and 10^{-6} for data quality applications. Thus, a bare minimum of 10^3 (resp. 10^6) information bits must be simulated to estimate a practical BER. All S⁴W work to date has dealt with voice quality applications. Empirically, these applications require 10^3 to 10^5 frames per wireless system configuration (each frame contains 80 information bits). The computational expense of these simulations is too high to perform them for all channels simulated by the ray tracer. Instead, we calculate statistics from the ray tracing output which are then used to generate a reasonable number of channels for the WCDMA simulations. Surrogate models are fit onto the WCDMA simulation outputs. The surrogate models are then compiled into the database functions. These functions can be used in subsequent queries in place of the WCDMA simulation outputs (see the following subsection for an example).

The `no_frames`, `max_frames`, `confidence`, `rel_error`, and `min_ber` columns define the stopping criteria for the simulation. These stopping criteria are described in detail in a latter chapter. Basically, batches of `no_frames` frames each are simulated until (a) we are confident that the relative error in the aggregate BER does not exceed `rel_error`, or (b) we are confident that the BER is below `min_ber`, or (c) the number of frames required to satisfy either of the previous tests exceeds `max_frames`. The `confidence` column gives the confidence levels for these tests. When the confidence level is zero, exactly one batch of `no_frames` frames is simulated. Otherwise, the stopping criteria are tested after each batch of frames, provided that at least two batches have been simulated.

The `wcdma_outputs` table contains the estimates of the BER and FER (frame error rate) of the wireless systems simulated. This table contains one record per batch of simulated frames. As the previous paragraph suggests, there can be more than one output record per input record. All such output records are aggregated into one by the view `wcdma_outputs_summary`. The aggregate results are simply the sample mean and sample variance of all BER (resp. FER) estimates per input record. A latter chapter illustrates the use of these values in a data mining study.

The WCDMA approach to statistically significant sampling can be viewed as an alternative to EMDAG's controller-executor relationship. The advantage of this approach is the simplicity of implementation. First, this approach does not require two separate components, a controller and an executor. Second, the code need not be generic. We can assume a particular distribution of outputs for hypothesis testing. Finally, the schemas of the executor and the controller are conveniently integrated. The disadvantage of this approach is

its lack of flexibility. E.g., sampling for high relative accuracy of the results is appropriate for surface fitting, but not for data mining. The following subsection illustrates the use of EMDAG's execution management to implement a more general-purpose iteration.

2.2.3 Numerical Optimization and Data Management

This subsection describes data management for VTDIRECT, a numerical optimizer implemented after the DIRECT algorithm of Jones *et al.* [39]. The facilities described here can be used to interface the VTDIRECT optimizer to any simulation component. We take a close look at VTDIRECT's interface to S⁴WRT and a MARS [26] surrogate model for the BER of a WCDMA system. The optimization study under consideration aims at finding the transmitter locations that yield an optimal BER coverage.

Numerical optimization can be viewed as an iterative process of the following form. An optimizer begins with an initial guess of the optimization variable values. At each iteration, an objective function is evaluated and the next set of variable values is constructed according to the optimization algorithm. This process stops when one of the optimization stopping criteria is satisfied.

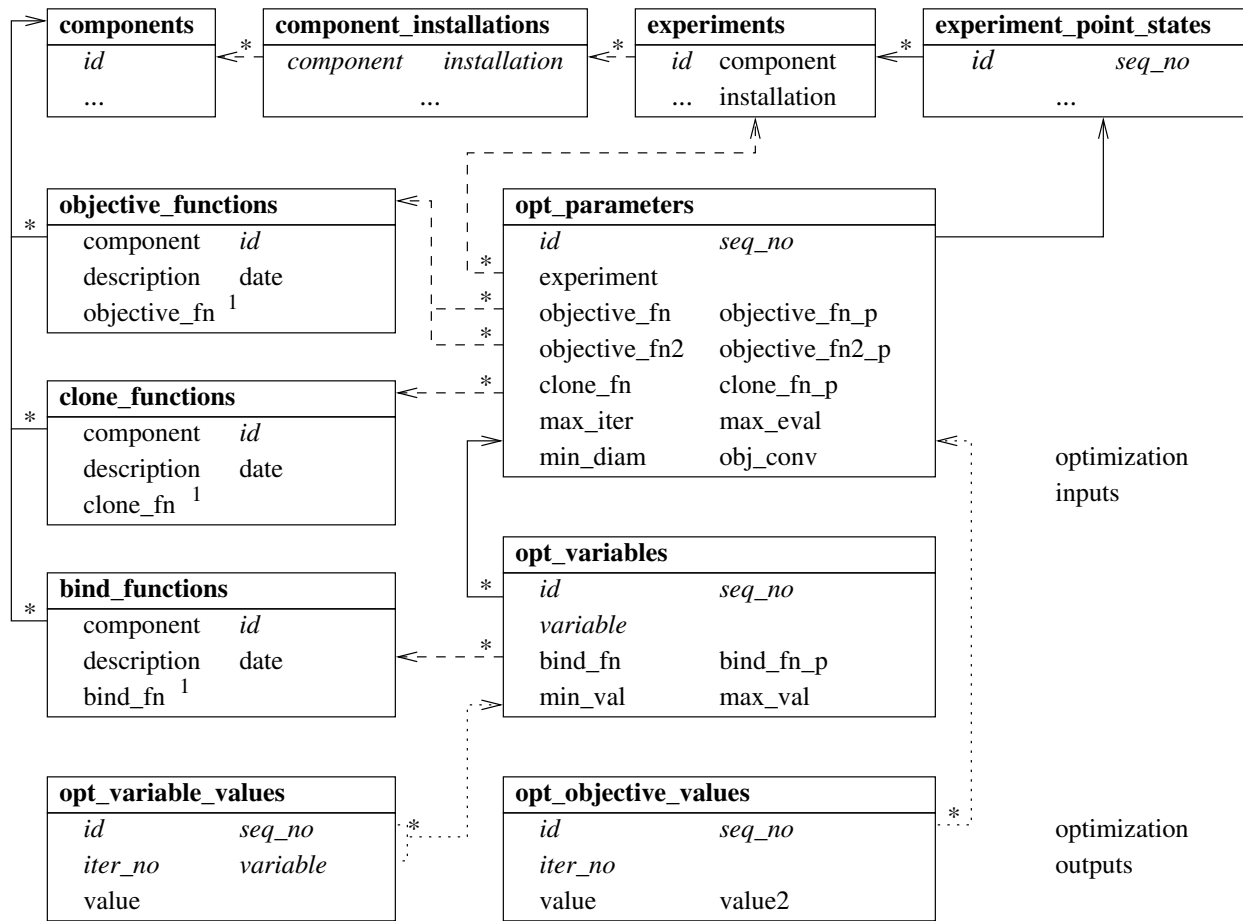
VTDIRECT Schemas

VTDIRECT schemas are shown in Figure 2.13. An optimization experiment consists of (a) one or two objective functions, (b) two or more optimization variables, (c) the 'slave' experiment used for objective function evaluation, and (d) a number of optimization parameters. The optimizer performs several objective function evaluations. For each evaluation, we store the values of the optimization variables and the objective function(s).

VTDIRECT schemas make heavy use of user-defined database functions. We distinguish between the three kinds of functions: the clone functions, the bind functions, and the objective functions. All these functions are usually implemented in SQL. The clone functions, one per optimization experiment, make copies of the slave experiment points. The bind functions, one per optimization variable, change the inputs of the slave experiment point to the values computed by the optimizer. Once the outputs of the slave experiment point have been computed, the objective functions, one or two per optimization experiment, compute aggregate measures of slave point 'goodness'. The first objective function drives the DIRECT algorithm. This function must be Lipschitz continuous [39]. The (optional) second objective function provides additional aggregate information for human consumption. This function need not be Lipschitz continuous.

Objective functions are well understood parameters to numerical optimization, but clone and bind functions are not so. Our use of clone and bind functions is based on the following observation. The optimizer should not be concerned with the fixed parameters of the slave experiment. The schemas of these parameters can be quite complicated (e.g., the ray tracing schemas in Figure 2.10 on page 23). We use the following trick to avoid the optimizer's involvement with these schemas. The user initially creates a template slave experiment point that contains the values of all fixed parameters. This point has a special sequence number -1 that is ignored by EMDAG's executors. The optimizer uses the clone function to make copies of this point and the bind functions to change the values of the parameters of interest. Thus, the clone and the bind functions effectively decouple the optimizer from the specifics of simulation schemas.

VTDIRECT is a serial program written in Fortran 95. A Tcl (Expect) wrapper script interfaces this program with the database and with EMDAG. EMDAG views the wrapper script as both a controller for the slave experiment and the executor for the optimization experiment.



¹: names of user-defined server (database) functions:
 objective_fn(id, seq_no, ...) -> value
 clone_fn(src_id, src_seq_no, dst_id, dst_seq_no, ...) -> boolean
 bind_fn(id, seq_no, value, ...) -> boolean

Figure 2.13: VTDIRECT schemas. SQL code for these schemas is available in Appendix B.3.

BER Coverage Optimization

Consider the user-defined functions in the BER coverage optimization. This particular optimization setup uses the ray tracer to estimate the impulse responses and a surrogate model to estimate the BERs. Refer to Figure 2.10 on page 23 to follow the SQL code presented here.

The clone function copies the inputs of the ray tracing point ($\$src_id, \src_seq_no) to the ray tracing point ($\$dst_id, \dst_seq_no). A fragment of this function is shown below.

```
INSERT INTO s4wrt_inputs (
    id, seq_no, octree_id, octree_seq_no,
    max_transmissions, transmission_loss, max_reflections, reflection_loss,
    min_power, power_range, path_loss_factor
) (SELECT
    $dst_id AS id, $dst_seq_no AS seq_no, octree_id, octree_seq_no,
    max_transmissions, transmission_loss, max_reflections, reflection_loss,
    min_power, power_range, path_loss_factor
FROM s4wrt_inputs
WHERE id = $src_id AND seq_no = $src_seq_no
);
INSERT INTO s4wrt_transmitter_locations (
    s4wrt_id, s4wrt_seq_no, id, seq_no,
    x, y, z, freq, power, tessl, refdist,
    antenna_id, antenna_seq_no, phi_a, theta_a
) (SELECT
    $dst_id AS s4wrt_id, $dst_seq_no AS s4wrt_seq_no, id, seq_no,
    x, y, z, freq, power, tessl, refdist,
    antenna_id, antenna_seq_no, phi_a, theta_a
FROM s4wrt_transmitter_locations
WHERE s4wrt_id = $src_id AND s4wrt_seq_no = $src_seq_no
);
...
```

The straightforward SQL code proceeds to copy all other ray tracing parameters.

The bind function for the X coordinate of the transmitter location assigns the value $\$value$ to the X coordinate of the transmitter location identified by $\$txloc_id$ in the ray tracing point ($\$id, \seq_no). This function is a single SQL query.

```
UPDATE s4wrt_transmitter_locations SET x = $value
WHERE s4wrt_id = $id AND s4wrt_seq_no = $seq_no AND
    ($txloc_id = '' OR id = $txloc_id);
```

The bind function for the Y coordinate is similar. Consider optimizing placement of multiple transmitters. We do not wish to write a separate bind function per transmitter location. Instead, the generic bind function takes an extra argument that selects the transmitter location to update. In general, clone, bind, and objective functions can take any number of extra arguments. The particular arguments are stored in the appropriate optimization input tables as Tcl lists. E.g., the extra arguments to the bind functions are stored in the `bind_fn_p` column of the `opt_variables` table.

The most creative part of the optimization experiment definition is writing the objective function. The aim of the objective function is to aggregate all experiment outputs into a single number. This number must be a meaningful performance metric. For BER coverage optimization, we are interested in the fraction of points that exhibit acceptable performance. (This objective function is not Lipschitz continuous, so DIRECT uses a smooth indicator function defined in [50].) We say that a point exhibits acceptable performance when the BER at this point is below a fixed threshold $\$T$. Let $\$D$ be the bandwidth resolution of the WCDMA system (in ns), $\$N$ be the receiver noise level (in dBW), $\$C$ be the dynamic range of the impulse response (1.0 to remove all but the peak power, 0.0 to retain all powers), and $(\$id, \$seq_no)$ be the ray tracing point of interest. Omitting some arcane PostgreSQL syntax, the objective function can be written as follows.

```
SELECT AVG(
  CASE WHEN wcdma_ber_rake(c.pdp,pow(10,$N/10)) < $T THEN 1.0
  ELSE 0.0
  END
) FROM (SELECT MAX(b.pdp) AS pdp FROM
  (SELECT txloc_id, rxloc_g_id, row, col, AVG(a.pdp) AS pdp FROM
    (SELECT
      txloc_id, txloc_ant_no, rxloc_g_id, row, col,
      AVG(trim_bins(resample(pdp,$D),$C)) AS pdp
    FROM s4wrt_rxloc_grid_pdps
    WHERE s4wrt_id = $id AND s4wrt_seq_no = $seq_no
    GROUP BY txloc_id, txloc_ant_no, rxloc_g_id, row, col) a
  GROUP BY txloc_id, rxloc_g_id, row, col) b
  GROUP BY rxloc_g_id, row, col) c;
```

The inner-most subquery (a) resamples the ‘raw’ impulse responses w.r.t. the bin width and the pulse width $\$D$, (b) removes the weak bins w.r.t. the threshold $\$C$, and (c) computes the average impulse response over all receiver location antennas. The impulse responses are averaged because the WCDMA simulation accepts only one impulse response for all receiver location antennas. Impulse response averaging is implemented as a user-defined database aggregate. The middle subquery (d) performs impulse response averaging over the transmitter location antennas. The surrogate model does not include the effects of transmit diversity. Therefore, experiments with diversity will have to settle for a conservative one-antenna bound on system performance. The outer-most subquery (e) selects the impulse response with the maximum peak power for each receiver location. We assume that each receiver selects the transmitter with maximum power. We also ignore the effects of inter-cell interference because the WCDMA simulation does not presently support it. Selection of the maximum impulse response is also implemented as a user-defined database aggregate. Finally, the whole query (f) computes the fraction of receiver locations whose BER is below the threshold $\$T$. The `wcdma_ber_rake()` surrogate function (C, FORTRAN) computes the BER and the standard SQL facilities do the rest.

To summarize, EMDAG’s controller-executor paradigm can be used to wrap optimization loops around arbitrary simulation components. Generality of the interface is achieved by an extensive use of custom database functions. Several such functions have been presented for a non-trivial optimization experiment. All of these functions are relatively simple SQL queries.

2.3 Discussion of Experiment Management

This chapter has demonstrated one set of data-centric and computation-centric facilities that, when combined, provide a reasonable skeleton for a problem solving environment. We have identified three classes of functionality (installation management, execution management, and data management) that form a useful experiment management system for a PSE. The concepts developed here have been validated in S⁴W, a PSE that contains fairly complex high-fidelity simulations of wireless channels and systems.

The relational data model used throughout this chapter is often dismissed in favor of more flexible data models. E.g., the following chapter assumes a semistructured data model and [36] assumes an object oriented data model. Nevertheless, we have shown that appropriate use of the relational model and a state-of-the-art database management system can provide reasonably high-level problem solving facilities. E.g., objective function evaluation for numerical optimization can be implemented via simple SQL queries. Likewise, format conversions and surrogate models for PSE data are easily implemented as database functions.

A sufficiently flexible relational database (e.g., PostgreSQL) can be viewed as an interpreter for a scripting language specifically designed for data manipulation. A library of custom data types, functions, and aggregation operators naturally specializes this language for a particular application domain. A number of such extensions have been presented for common operations in wireless system design (e.g., impulse response averaging). The utility of this overloading of the query semantics is potentially enormous. Not only does this approach drastically simplify format conversion of simulation data, it can ultimately change the way the users look at their experiments. Parts of simulations can, in principle, be implemented as database queries. Pushing these queries as deep as possible [34] into the simulation is a worthy direction for future research.

On a more practical note, the overwhelming advantages of the relational model over its more flexible competition are the maturity of the relational model and the wide availability of high quality relational databases. Neither the semistructured nor the object oriented data model can currently compete with the relational model along these dimensions.

The present work can be extended in a large number of directions. This chapter covered a relatively simple system that provides only the most essential functionality. Different kinds of functionality can be built on top of EMDAG.

For instance, the current version of EMDAG does not have a security policy or access controls. Any user can change any table, including the tables that contain commands invoked on other machines. A security policy can be implemented via triggers but the complicated part is not implementing the policy; it is defining a policy suitable for use in a PSE.

Likewise, some projects may benefit from a general-purpose workflow management system. Relational schemas enforce some constraints on workflow, but there is no centralized workflow management. Workflows can be scripted (e.g., in Tcl), but there is a long way from scripting workflows in a general-purpose language to a high-level workflow management system.

Finally, high-level domain-specific schema editing tools can be built. However, one must realize that databases do not magically solve all data management problems. No amount of point-n-click gadgets can replace a careful schema design. Designing appropriate schemas and integrating these schemas with those of the other system components is ultimately the responsibility of the simulation developer. The most problematic part of schema development is the precise definition of simulation inputs and outputs; once the simulation is well defined, the data model follows.

Chapter 3

BSML: A Binding Schema Markup Language for Data Interchange in Problem Solving Environments

This chapter is devoted to a binding schema markup language (BSML) for describing data interchange between scientific codes. Such a facility is an important constituent of scientific PSEs. BSML is designed to integrate with a PSE or application composition system that views model specification and execution as a problem of managing semistructured data. The data interchange problem is addressed by three techniques for processing semistructured data: validation, binding, and conversion. We present BSML and describe its application to S⁴W.

We view model specification and execution as a data management problem and describe how a semistructured data model can be used to address data interchange problems in a PSE. Section 3.1 presents a motivating S⁴W scenario that will help articulate the needs from a data management perspective. Section 3.2 elaborates on these ideas and briefly reviews pertinent related work. In particular, it identifies three basic levels of functionality—validation, binding, and conversion—at which data interchange in application composition can be studied. Sections 3.3, 3.4, and 3.5 describe our specific contributions along these dimensions, in the form of a binding schema markup language (BSML). Section 3.6 outlines how these ideas can be integrated within an existing PSE system design. A concluding discussion is provided in Section 3.7. Aspects of the scenario described next will be used throughout this chapter as running examples.

3.1 Motivating Example

Consider the following usage scenario for S⁴W. A wireless design engineer wishes to study transmitter placement in an indoor environment located on the fourth floor of Durham Hall at Virginia Tech. The engineering goal is to achieve a certain performance objective within the given cost constraints. For a narrowband system, power levels at the receiver locations are good indicators of system performance. Therefore, minimizing the (spatial) average shortfall of received power with respect to some power threshold is a meaningful and well defined objective. The major cost constraints are the number of transmitters and their powers. Different transmitter locations and powers yield different levels of coverage. The situation is more complicated in a wideband system, but roughly the same process applies. A wideband system includes extra hardware not present in a narrowband system and the performance objective is formulated in terms of the bit error rate (BER), not just the power level.

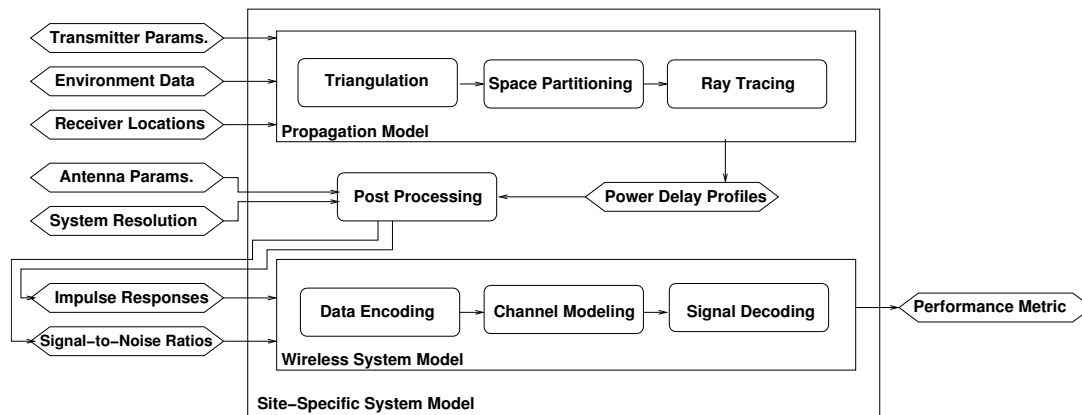


Figure 3.1: A site-specific system model in S⁴W. The system model consists of a propagation model, an antenna model (post processing), and a wireless system model.

The first step in this scenario is to construct a model of signal propagation through the wireless communications channel. S⁴W provides ray tracing as the primary mechanism to model site-specific propagation effects such as transmission (penetration), reflection, and diffraction. The second step is to take into account antenna parameters and system resolution. These two steps are often sufficient to model the performance of a narrowband system. If a wideband system is being considered, the third step is to configure the specific wireless system. Parameters such as the number of fingers of the rake receiver and forward error correction codes are considered at this step. S⁴W provides a Monte-Carlo simulation of a WCDMA (wideband code division multiple access) family of wireless systems. In either case, the engineer configures a graph of computational components as shown in Fig. 3.1. The ovals correspond to computational components drawn from a mix of languages and environments. Hexagons enclose input and output data. Aggregation is used to simplify the interfaces of the components to each other and to the optimizer. In Fig. 3.1, rectangles represent aggregation. The propagation model is a component that consists of three connected sub-components: triangulation, space partitioning, and ray tracing. Similarly, the wireless system model consists of (roughly) three components: data encoding, channel modeling, and signal decoding. All three steps are further aggregated into a complete site-specific system model. This model is then used in an optimization loop. The optimizer changes transmitter parameters (all other parameters remain fixed) and receives feedback on system performance.

For a given environment definition in AutoCAD, the triangulation and space partitioning components are used to reduce the number of geometric intersection tests that will be performed by the ray tracer. Several iterations over space partitioning are necessary to achieve acceptable software performance. However, once the objective (an average of ten triangles per voxel) is met, the space partitioning can be reused in all future experiments with this environment. The engineer then configures the ray tracer to only capture reflection and transmission (penetration) effects. Although diffraction and scattering are important in indoor propagation [7], these phenomena are computationally expensive to model in an optimization loop. The triangulation and space partitioning codes are meant for serial execution, whereas the ray tracer and the Monte Carlo wireless system models run on a 200 node Beowulf cluster of workstations. Post processing is available in both serial and parallel versions. The ray tracer and the post processor are written in C, whereas the WCDMA simulation is available in Matlab and Fortran 95 versions.

A series of experiments is performed for various choices of antenna patterns, path loss parameters (influenced by material properties), and WCDMA system parameters. The predicted power delay profiles (PDPs)

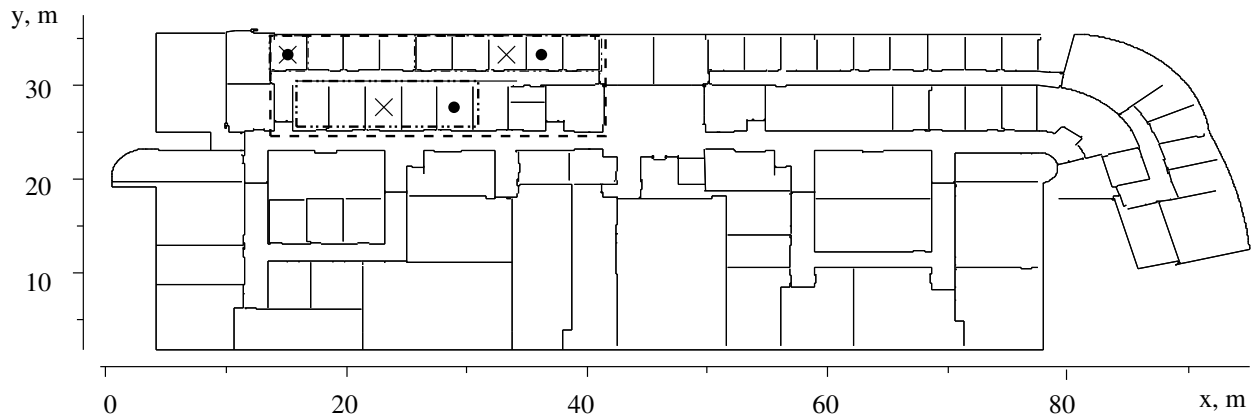


Figure 3.2: Optimizing placement of three transmitters to cover eighteen rooms and a corridor bounded by the box in the upper left corner. The bounds for the placement of three transmitters are drawn with dotted lines. The initial transmitter positions are marked with crosses. The optimum coverage transmitter positions are marked with dots.

are then compared with the measurements from a channel sounder and the predicted bit error rates are compared with the published data. The parameters of the propagation model are calibrated for various locations. The validated propagation and wireless system models are finally enclosed in an optimization loop to determine the locations of transmitters that will provide adequate performance for a region of interest. The optimizer, written in Fortran 95, uses the DIviding RECTangles (DIRECT) algorithm of Jones et al. [39]. The parameters to the optimization problem and the optimal transmitter placement are depicted in Fig. 3.2. The optimizer decided to move the transmitter in the upper right corner one room to the right of its initial position and the transmitter in the lower left corner two rooms to the right of its initial position.

What requirements can we abstract from this scenario and how can they be flexibly supported by a data model? We first observe the diversity in the computational environment. Component codes are written in different languages and some of them are meant for parallel execution. In a research project such as S⁴W, many components are under active development, so their I/O specifications change over time. Second, the interconnection among components is also flexible. Optimizing for power coverage and optimizing for bit error rate, while having similar motivations, require different topologies of computational components. Third, since different groups of researchers are involved in the project, there exists significant cognitive discordance among vocabularies, data formats, components, and even methodologies. For example, ray tracing models represent powers in a power delay profile in dBm (log scale). However, WCDMA models work with a normalized linear scale impulse response and an aggregate called the ‘signal-to-noise ratio.’ Also, there is more than one way of calculating the energy-to-noise ratio. Since antennas generate noise that depends on their parameters, detailed antenna descriptions are necessary to calculate this ratio. However, researchers who are not concerned with antenna design seldom model the system at this level of detail. The typical practice is to use a fixed noise level in the calculations. Simulations of wireless systems abound in such approximations, ad hoc conversions, and simplifying assumptions.

3.2 PSE Requirements for Data Interchange

Culling from the above scenario, we arrive at a more formal list of data interchange requirements for application composition in a PSE. The PSE must support:

1. components in multiple languages (C, FORTRAN, Matlab, SQL);
2. changes in component interfaces;
3. changes in interconnections among components;
4. automatic unit conversion in data-flows;
5. user-defined conversion filters;
6. composition of components with slightly different interfaces; and
7. stream processing.

The reader might be surprised that SQL is listed alongside FORTRAN, but both languages are used in S⁴W. Experiment simulations are written in procedural languages, while experiment data is stored in a relational database. Thus, developing a system that integrates with the PSE environment requires more than the ability to link scientific computing languages. It involves overcoming the impedance mismatch between languages developed for fundamentally different purposes.

Stream processing refers to processing data as soon as it is read from the input stream, as opposed to waiting for the end of the stream and then processing all the data at once. This often neglected technical requirement is related to composability—the ability to create arbitrary component topologies. As data interchange is pushed deeper into the computation, the unit of data granularity needs to become correspondingly smaller. The optimization loop is a good example of fine data granularity. We cannot accumulate all transmitter parameters over all iterations and later convert them to the format required by the simulation inside the loop, because transmitter parameters generated by the optimizer depend on the feedback computed by the simulation. Each block of transmitters must be processed as soon as it is available. Likewise, each value of the objective function must be made available to the optimizer before it can produce the next block of transmitters. Usability dictates a similar requirement. Since some models are computationally expensive (e.g., those meant for parallel execution), incremental feedback should be provided to the user as early as possible. The stream processing requirement improves composability and usability, but limits conversions to being local. Global conversions (e.g., XSLT [19]) cannot be performed because they assume that all the data is available at once.

While the requirements point to a semistructured data model, no currently available data management system supports all forms of PSE functionality. This chapter presents the prototype of such a system in the form of a markup language. Observe that all of the above requirements are summarized by three standard techniques for working with semistructured data—validation, binding, and conversion. *Validation* establishes data conformance to a given schema. It is a prerequisite to most of the requirements. *Binding* refers to integrating semistructured data with languages that were designed for different purposes (requirement 1). *Conversion* (transformation) takes care of requirements 2–6. Given two slightly different schemas, it is possible to generate an *edit script* [17] that converts data instances from one schema to another. Requirement 7 dictates that all such conversions must be local.

3.2.1 Related Work

While research in PSEs covers a broad territory, the use of semistructured data representations in computational science is not established beyond a few projects. Therefore, we only survey standard XML technologies and PSE-like systems that make (some) use of semistructured data. It would be unfair to review some

of these systems against PSE data interchange requirements. Instead, our evaluation is based on how well these systems support validation, binding, conversion, and stream processing.

Specific XML technologies for document processing are easy to classify in terms of our framework. *Schema languages* (e.g., RELAX NG [18]) deal with validation and, possibly, binding. *Transformation languages* (e.g., XSLT [19]) deal with conversion. Several properties of these technologies hinder their direct applicability to a PSE setting. First and foremost, these technologies do not work with streams of data. Sophisticated schema constraints and complex transformations can require buffering the whole document before producing any output. Second, transformation languages are simply vehicles for applying edit scripts. They cannot be used to create edit scripts. Since our conversions are local, edit script application is trivial, but edit script creation is not.

Four major flavors of PSE-like projects that use semistructured data representations can be identified:

1. component metadata projects;
2. workflow projects;
3. scientific data interchange projects; and
4. scientific data management projects.

Projects in the first category use XML to store IDL-like (interface definition language) component descriptions and miscellaneous component execution parameters. An example of such a project is CCAT [13], which is a distributed object oriented system. CCAT also uses XML for message transport between components, so we say that it provides an OO binding. The second category of projects augments component metadata with workflow specifications. For example, GALE [12] is a workflow specification language for executing simulations on distributed systems. Unlike CCAT, GALE provides XML specifications for some common types of experiments, such as parameter sweeps (CCAT uses a scripting language for workflow specification). However, GALE does not use XML for component data. Both the component metadata and workflow projects use XML to encode data that is not semistructured. Their use of XML is not dictated by the need for automatic conversion. Neither generic binding mechanisms nor conversion are provided by these projects.

The latter two groups of projects use XML for application data, not component metadata. Representatives of the scientific data interchange group develop flexible all-encompassing schemas for specific application domains. For example, CACTUS [11] deals with spatial grid data. CACTUS's schema is complex enough to be considered semistructured and this project recognizes the need for conversion filters. However, it does not provide multiple language support and, more importantly, does not accommodate changes in the schema. CACTUS's conversion filters aim at code reuse, not change management. This project has OO binding and manual conversion (the sequence of conversions is not determined automatically). Complexity of the data format precludes stream processing.

Perhaps the most relevant group of projects for our purposes involves the scientific data management community. Especially interesting are the projects in rapidly evolving domains, such as bioinformatics. DataFoundry [1, 20] provides a unifying database interface to diverse bioinformatics sources. Both the data and the schema of these sources evolve quickly, so DataFoundry has to deal with change management—by far more complex change management than the kind we consider here. However, DataFoundry only provides *mediators* for database access. It does not integrate with simulation execution. This system takes full advantage of conversion, but provides only an SQL binding. Introducing bindings for procedural languages would involve significant changes to DataFoundry.

Table 3.1 summarizes related work. It turns out that no known PSE-like system takes full advantage of both binding and conversion. XML technologies for validation and binding are well established, but XML transformation technologies do not support PSE-style conversion. Very few systems can integrate with a

	CCAT	GALE	CACTUS	DataFoundry	RELAX NG	XSLT
Validation	✓		✓	✓	✓	
Binding	OO		OO	SQL	OO	
Conversion			manual	✓		manual
Stream Processing	✓					

Table 3.1: A survey of PSE-like systems and XML technologies. The binding row shows that most systems support only one paradigm. Only DataFoundry fully supports conversion. Other systems either provide a library of conversion primitives and leave their composition up to the user (CACTUS) or do not recognize the need for conversion at all (CCAT). No system or technology fully supports validation, binding, and conversion. Most systems and technologies cannot dynamically process streams of data.

PSE execution environment because most of them do not meet the stream processing requirement. This chapter develops a system that satisfies all of our data interchange requirements. The next three sections describe our handling of validation, binding, and conversion. System integration is outlined in Section 3.6.

3.3 Validation

Validation establishes conformance of a data instance to a given schema. It is a prerequisite to binding and conversion. (This definition of validation is a small part of the process of validation in a PSE, which is concerned with the larger issue of a model being appropriate to solve a given problem; but, it suffices for the purpose of this chapter.) The schemas for PSE data are easy to obtain since computational science traditionally uses rigid data structures, not loosely formatted documents. Describing the data structures in terms of schemas has several benefits. First, language-neutral schemas allow for interoperability between different languages (see requirement 1 in the previous section). Second, schemas facilitate database storage and retrieval. Third, appropriate schemas help assign interpretations to various data fields. It is such interpretation that makes automatic conversion possible (requirements 2–6).

What kind of validation is appropriate for PSE data? Requirement 7 calls for the most expressive schema language that can be parsed by a stream parser. In other words, we are looking for a schema language that can be defined in terms of an LL(1) grammar [3]. (The LR family of grammars is more expressive, but LR parsers do not follow stream semantics.) Therefore, a predictive parser generated for a given schema can validate a data instance. This section describes a schema language (BSML) appropriate for a PSE and the steps for building a parser generator for this language. We present an example, an informal overview of BSML features, and a formal definition for a large subset of BSML in terms of a context-free grammar. Further, predictive parser generation is outlined and grammar transformations specific to BSML are described in detail. Finally, we show that BSML is strictly less expressive than LL(1) grammars.

Let us start with an example. Figures 3.3 and 3.4 depict a (simplified) schema for an octree environment decomposition. (Fig. 3.3 describes it in XML notation while Fig. 3.4 uses a non-XML format that will be useful for describing some functionalities of BSML). This was the most complex schema in S⁴W, not counting the schema for the schema language itself. An octree consists of internal and leaf nodes that delimit groups of triangles. Recall from Section 3.1 that this grouping is used to limit the intersection tests in ray tracing. The nested structure of an octree maps nicely into an XML tree. Since many components work with lists of triangles, there is a separate schema for a list of triangles. As the example shows, the features of BSML closely resemble those of other schema languages, such as RELAX NG. The only noticeable difference is the presence of units in the definitions of primitive types. Units will be useful for certain types

of conversions. Figure 3.5 shows an LL(1) grammar generated from the octree schema. This grammar is then annotated with binding code and used to generate a parser for octree data. The parser can be linked with a parallel ray tracer written in C.

The DTD for the current version of BSML is given in Appendix C. The schema language describes primitive types and schemas. There are four base primitive types: integer, string, (IEEE) double, and boolean. Users can derive their own primitive types by range restriction. User-derived types usually have domain-specific flavor, such as coordinates and distances in the example above. We do not support more complicated primitive types, such as dates and lists, because each PSE component treats them differently. Schemas consist of four building blocks: elements, sequences, selections, and repetitions. Strictly speaking, repetitions can be expressed as selections and sequences, but they are so common that they deserve special treatment. Derivation of schemas by restriction is not supported, but derivation by extension can be implemented via inter-schema references. Mixed content is not supported because it is only used for documentation. Instead, BSML supports a wildcard content type. The contents of this type matches anything and is delivered to the component as a DOM tree [9]. We do not support referential integrity constraints because they can delay binding and thus break requirement 7. There is no explicit construct for interleaves. In some ways, interleaves are handled by the conversion algorithm. In other words, BSML is a simple schema language that incorporates most common features that are useful in a PSE.

Parser generation for a BSML schema follows the standard steps from compiler textbooks [3]:

1. convert the schema to an LL(1) grammar,
2. eliminate empty productions and self-derivations,
3. eliminate left recursion,
4. perform left factoring,
5. perform miscellaneous cleanup (described in detail below),
6. compute a predictive parsing table, and
7. generate parsing code from the table.

The only steps specific to this schema language are generating an LL(1) grammar (step 1) and miscellaneous cleanup (step 5). Since grammars have been in use for a long time, it is pertinent to define BSML semantics in terms of how the schemas are converted to grammars. The terminals are defined by SAX events [15]. The start of element and end of element events are denoted $s(name)$ and $e(name)$, respectively, where $name$ is element name. We omit the attributes for simplicity, but BSML supports them in an obvious way. Further, we assume that the SAX parser inlines external entity references. Character data is accumulated until the next start of element or end of element event and delivered as a $d(base, min, max, number, finite, units)$ terminal, abbreviated as d (see Appendix C for d 's attributes). Generated code checks character data conformance to the type constraints. This definition of d is appropriate since BSML does not support selections based on the type of character data.

One root non-terminal is initially generated for each schema block (element, sequence, selection, repetition), each reference to a primitive type, and each string of user code. We denote non-terminals by capital letters, the start non-terminal by S , the empty string by ϵ , and the root non-terminals generated for the children of each schema block by $X_1, X_2, \dots, X_n, n \geq 0$. Further, lower-case Greek letters denote (possibly empty) sequences of terminals, non-terminals, and, in the next section, user codes. With this notation in mind, the definition of BSML is in Figure 3.6 (more details follow in future sections). We slightly deviate from a context-free grammar to allow for the constraints on the number of repetitions (see next section). To reiterate, a grammar generated from a schema according to this definition will undergo several standard equivalence transformations before a grammar of the form shown in Figure 3.5 is obtained.

```

<type id='distance' base='double' number='true' finite='true' />
<type id='coordinate' base='double' number='true' finite='true' />

<schema id='triangles'>
  <repetition>
    <element name='tr'>
      <repetition min='3' max='3'>
        <element name='v'>
          <attribute name='x' type='coordinate' units='m' />
          <attribute name='y' type='coordinate' units='m' />
          <attribute name='z' type='coordinate' units='m' />
        </element>
      </repetition>
    </element>
  </repetition>
</schema>

<schema id='octree'>
  <element name='octree'>
    <element name='oi' id='oi'>
      <attribute name='x' type='coordinate' units='m' />
      <attribute name='y' type='coordinate' units='m' />
      <attribute name='z' type='coordinate' units='m' />
      <attribute name='dx' type='distance' units='m' />
      <attribute name='dy' type='distance' units='m' />
      <attribute name='dz' type='distance' units='m' />
      <ref id='triangles' />
      <repetition>
        <selection>
          <ref id='oi' />
          <element name='ol'>
            <attribute name='x' type='coordinate' units='m' />
            <attribute name='y' type='coordinate' units='m' />
            <attribute name='z' type='coordinate' units='m' />
            <attribute name='dx' type='distance' units='m' />
            <attribute name='dy' type='distance' units='m' />
            <attribute name='dz' type='distance' units='m' />
            <ref id='triangles' />
          </element>
        </selection>
      </repetition>
    </element>
  </element>
</schema>

```

Figure 3.3: BSML schemas for an octree decomposition of an environment, in XML notation. ‘tr’ stands for a triangle, ‘v’ stands for a vertex, ‘oi’ stands for an internal node, and ‘ol’ stands for a leaf.

```

type(distance, double, $, $, true, true, $)
type(coordinate, double, $, $, true, true, $)

schema(triangles,
  repetition($, $, $, $,
    element($, $, tr,
      repetition($, $, 3, 3,
        element($, $, v,
          attribute($, x, data(coordinate,$,$,$,$,m)),
          attribute($, y, data(coordinate,$,$,$,$,m)),
          attribute($, z, data(coordinate,$,$,$,$,m))
        )
      )
    )
  )
)

schema(octree,
  element($, $, octree,
    element(oi, $, oi,
      attribute($, x, data(coordinate,$,$,$,$,m)),
      attribute($, y, data(coordinate,$,$,$,$,m)),
      attribute($, z, data(coordinate,$,$,$,$,m)),
      attribute($, dx, data(coordinate,$,$,$,$,m)),
      attribute($, dy, data(coordinate,$,$,$,$,m)),
      attribute($, dz, data(coordinate,$,$,$,$,m)),
      ref(triangles),
      repetition($, $, $, $,
        selection($, $,
          ref(oi),
          element($, $, ol,
            attribute($, x, data(coordinate,$,$,$,$,m)),
            attribute($, y, data(coordinate,$,$,$,$,m)),
            attribute($, z, data(coordinate,$,$,$,$,m)),
            attribute($, dx, data(coordinate,$,$,$,$,m)),
            attribute($, dy, data(coordinate,$,$,$,$,m)),
            attribute($, dz, data(coordinate,$,$,$,$,m)),
            ref(triangles)
          )
        )
      )
    )
  )
)
)

```

Figure 3.4: BSML schemas from Figure 3.3 in a non-XML notation. \$ stands for a missing value, i.e., a suitable default value is supplied by BSML software.

S	\rightarrow	$s(\text{octree}), s(\text{oi}), T, C, e(\text{oi}), e(\text{octree})$
T	\rightarrow	ϵ
T	\rightarrow	$\{B_t\}, s(\text{tr}), \{B_v\}, s(v), e(v), \{A_v\}, V, \{E_v\}, e(\text{tr}), \{A_t\}, T', \{E_t\}$
T'	\rightarrow	ϵ
T'	\rightarrow	$s(\text{tr}), \{B_v\}, s(v), e(v), \{A_v\}, V, \{E_v\}, e(\text{tr}), \{A_t\}, T'$
V	\rightarrow	ϵ
V	\rightarrow	$s(v), e(v), \{A_v\}, V$
C	\rightarrow	ϵ
C	\rightarrow	$\{B_i\}, C', \{A_i\}, C'', \{E_i\}$
C'	\rightarrow	$s(\text{oi}), T, C, e(\text{oi})$
C'	\rightarrow	$s(\text{ol}), T, e(\text{ol})$
C''	\rightarrow	ϵ
C''	\rightarrow	I
I	\rightarrow	$s(\text{oi}), T, I'$
I	\rightarrow	$s(\text{ol}), T, e(\text{ol}), \{A_i\}, C''$
I'	\rightarrow	$\{B_i\}, C', \{A_i\}, C'', \{E_i\}, e(\text{oi}), \{A_i\}, C''$
I'	\rightarrow	$e(\text{oi}), \{A_i\}, C''$

Figure 3.5: LL(1) grammar corresponding to the octree schemas in Figures 3.3 and 3.4.

Attributes are omitted for simplicity. Patterns of the form $\{c\}$ will be explained in the next section (they are related to repetitions). Non-terminals T , T' , and V are related to triangles; others are related to octree decomposition of a set of triangles.

The purpose of miscellaneous cleanup is to reduce the number of non-terminals in the grammar. These ad-hoc rewritings do not guarantee that the resultant grammar is minimal in any strict sense. Instead, they address some inefficiencies that other steps are likely to introduce. These cleanup steps were also chosen such that if the grammar were LL(1) before cleanup, it would remain LL(1) after cleanup. The grammars shown in this chapter have undergone two cleanup rewritings. Each rewriting is applied until no further rewriting is possible.

1. Maximum length common suffixes are factored out. $\beta \neq \epsilon$ is the maximum length common suffix of a non-terminal $A \neq S$ if (a) all of A 's productions have the form $A \rightarrow \alpha_i\beta$, $1 \leq i \leq n$, (b) β is of maximum length, and (c) neither β nor any α_i contain A . If $n = 1$, A is eliminated from the grammar and all occurrences of A in the grammar are replaced with β ($\alpha_1 = \epsilon$ because β is of maximum length). We call such non-terminals trivial. Trivial non-terminals are often introduced by schema-to-grammar conversion rules. If $n > 1$, all occurrences of A on the right-hand sides of all grammar productions are replaced with $A\beta$ and the suffix β is deleted from all of A 's productions. The purpose of this rewriting is to uncover duplicate non-terminals for the next step.
2. Only one of any two duplicate non-terminals is retained. Two non-terminals $A \neq B$ are duplicate if whenever $A \rightarrow \alpha$ is in the grammar, $B \rightarrow \alpha$ is also in the grammar, and vice versa. A is eliminated if $A \neq S$, B is eliminated otherwise. This definition is weak, e.g., A and B are not considered duplicate if $A \rightarrow \alpha A\beta$ and $B \rightarrow \alpha B\beta$ are in the grammar. However, it suffices for our purposes.

The expressive power of LL(1) grammars is well known. In practice, the limiting factor is not that the grammar is LL(1), but that the grammar is annotated with user codes. The next section gives two examples

$\text{element}(id, opt, name, B_1, B_2, \dots, B_n)$	$E \rightarrow s(name), X_1, X_2, \dots, X_n, e(name)$
	$E \rightarrow \epsilon$ if opt
$\text{sequence}(id, opt, B_1, B_2, \dots, B_n)$	$Q \rightarrow X_1, X_2, \dots, X_n$
	$Q \rightarrow \epsilon$ if opt
$\text{selection}(id, opt, B_1, B_2, \dots, B_n)$	$L \rightarrow X_1$
	$L \rightarrow X_2$
	\dots
	$L \rightarrow X_n$
	$L \rightarrow \epsilon$ if opt
$\text{repetition}(id, opt, min, max, B_1, B_2, \dots, B_n)$	$R \rightarrow \{B\}, X_1, X_2, \dots, X_n, \{A\}, R', \{E\}$
	$R' \rightarrow X_1, X_2, \dots, X_n, \{A\}, R'$
	$R' \rightarrow \epsilon$
	$R \rightarrow \epsilon$ if opt or $min = 0$
$\text{data}(base, min, max, number, finite, units)$	$D \rightarrow d(base, min, max, number, finite, units)$
$\text{code}(c)$	$C \rightarrow \{c\}$

Figure 3.6: L-attributed definition of BSML. Schema primitives, in a non-XML notation, are on the left (see Figure 3.4 for an example) and their translations to grammar productions are on the right. B_1, B_2, \dots, B_n are the children of the schema block and X_1, X_2, \dots, X_n are the root non-terminals generated for B_1, B_2, \dots, B_n , respectively. opt is a boolean block attribute; true means that the block is optional. $\{B\}$, $\{A\}$, $\{E\}$, and $\{c\}$ are binding codes explained in the next section. References to schema blocks (denoted by $\text{ref}(id)$) are replaced with root non-terminals of the blocks being referenced. Definitions related to XML attributes are omitted.

of grammars that are not convertible to LL(1) because binding codes are present. A more interesting question is how the expressive power of LL(1) grammars compares to the expressive power of BSML. It is easy to see that BSML can express a proper subset of LL(1) grammars. For example, $S \rightarrow s(x), e(y)$ is a valid LL(1) grammar, but BSML cannot express it since no XML document that conforms to this grammar is well-formed.

Observation 1. Consider a subset of BSML that excludes repetitions and user codes. We say that BSML can express a grammar G if a predictive parser generated from some schema in this restricted subset of BSML can recognize precisely the language $L(G)$. Clearly, BSML cannot express any grammar G that is not LL(1) (by construction of the predictive parser). Further, BSML cannot express an LL(1) grammar G unless:

1. if d_1 and d_2 are data terminals in G , then $\forall \alpha, \beta : S \Rightarrow^+ \alpha, d_1, d_2, \beta$ (data is atomic),
2. if d is a data terminal and $S \Rightarrow^+ \alpha, d, \beta$ is a derivation in G , then $\forall x, \gamma : \left([\beta \Rightarrow^* s(x), \gamma] \text{ and } [(\beta \Rightarrow^* e(x), \gamma) \text{ implies } (\forall y, \theta : \alpha \Rightarrow^* \theta, e(y))] \right)$ (no mixed contents), and
3. if $s(x)$ is a start of element terminal, g is ϵ or a data terminal, and $S \Rightarrow^+ \alpha, s(x), \beta$ is a derivation in G , then $\left([\beta \Rightarrow^* g] \text{ and } [(y \neq x) \text{ implies } (\forall \gamma : \beta \Rightarrow^* g, e(y), \gamma)] \right)$; similarly, if $e(y)$ is an end of element terminal and $S \Rightarrow^+ \alpha, e(x), \beta$ is a derivation in G , then $\left([\alpha \Rightarrow^* g] \text{ and } [(x \neq y) \text{ implies } (\forall \theta : \alpha \Rightarrow^* \theta, s(x), g)] \right)$ (proper nesting of elements). □

The first two restrictions are specific to BSML and easy to relax. However, the last restriction is inherent in any XML schema language. A good schema language cannot describe documents that are not well-formed. These are the necessary conditions, but it is not clear whether or not they are sufficient. We define schemas in terms of the schema language, not in terms of LL(1) grammars, so converting from grammars to schemas is not considered in this chapter.

This section provided an overview of BSML features and defined BSML in terms of an ‘almost context-free’ grammar. We outlined automatic generation of predictive parsers that validate XML documents. Further, we have shown that the descriptive power of BSML is strictly less than that of an LL(1) grammar where the terminals are SAX events. The next section extends validation to perform binding.

3.4 Binding

Binding is a way to integrate semistructured data with languages that were not designed to handle it (requirement 1). Binding can take several forms, depending on the language. For FORTRAN and C, binding usually means assigning values to language variables and calling user-defined code to process these values (procedural binding). It can also mean writing the data out in a format understood by the component (format conversion). For Matlab and SQL, binding entails generating a script that contains embedded data and processing this script by an interpreter (code generation). The last two kinds of binding can be thought of as XSLT-like transformations.

We implement all three kinds of binding by L-attributed definitions. The schema language is extended by allowing user code to be injected in the schema. Schema languages that provide binding are called *binding schema markup languages*. This section describes bindings in BSML and gives an example of

their use. Further, we show how arbitrary binding codes limit the set of schemas supported by BSML. Predictive parsing cannot handle common prefixes in alternative productions, so standard techniques are used to eliminate such common prefixes. We show that these techniques break when the common prefixes contain binding codes. This limitation is rarely an issue and the problems it causes can be remedied by simple modifications to the schema.

Let c denote an arbitrary string of code. Matching $\{c\}$ means executing code c while consuming no input tokens. No assumptions are made about the nature of c . In particular, c can (and usually does) produce side effects, so $A \rightarrow \{c_1\}, \{c_2\}$ and $A \rightarrow \{c_2\}, \{c_1\}$ can yield different results. A *syntax-directed definition* is a context-free grammar extended by allowing $\{c_j\}$ on the right-hand sides of productions. For a syntax-directed definition to be useful in binding, c_j must contain references to parts of the document being parsed. We denote such references by $\%x$, where x is the id or the name of some element or attribute. When x refers to an attribute or an element of some primitive type, $\%x$ is a value of the attribute or the data contents of the element. The type of $\%x$ is determined by the corresponding primitive type. When x refers to an element of a wildcard type, $\%x$ is a DOM tree constructed from all descendants of x , including itself. This feature can be used for XHTML [41] documentation. The set of attributes (elements) that are available to code c depends on the placement of c in the syntax-directed definition and the parsing strategy. A syntax-directed definition is *L-attributed* if, for any derivation $S \Rightarrow^+ \alpha\{c\}\beta$, any x referenced in c is defined in all derivations of α . That is, all attributes (elements) must be defined in a left-to-right scan before they are referenced. L-attributed definitions are easy to implement with an LL(1) parser, but they restrict the set of grammars reducible to LL(1). Luckily, these restrictions are not important in practice.

Figure 3.7 gives an example binding schema for a PDP (see Section 3.1) and Figure 3.8 shows how a parser generated from this schema converts a PDP encoded in XML to a Matlab script. This script will then be executed by an execution manager (see Section 3.6). The same schema, with different binding code, can convert an XML file to a number of SQL INSERT statements that record the data in a relational database. The semantics of user codes are not limited to printing, so a FORTRAN version of this binding can store the PDP in an array to be processed later. In other words, BSML bindings are compatible with any execution environment that processes streams of data (requirement 7). We use the same approach to convert semistructured data to relational data, Matlab scripts, and C structures.

The $\{B\}$, $\{A\}$, and $\{E\}$ codes in Figure 3.7 are generated for repetitions. They are not necessary for this example, but are required to enforce that each triangle has three vertices in the previous example. $\{B\}$ (begin repetition) initializes the repetition count to zero. Each repetition has its own stack of counts. $\{A\}$ (append) ensures that the maximum allowed number of repetitions is not exceeded. $\{E\}$ (end) checks the minimum number of repetitions. Thus, even simple validation (without binding) is implemented in terms of an L-attributed definition, not just an LL(1) grammar.

Unfortunately, L-attributed definitions make predictive parsing of certain grammars impossible. User codes can prevent elimination of left recursion or left factoring of an L-attributed definition. In the two examples below, grammars induced from the left-attributed definitions by removing all user code can be transformed to LL(1). However, the original L-attributed definitions cannot be transformed to LL(1) without losing the stream semantics of the parser.

Example 1. Consider a left-recursive schema and the corresponding left-recursive grammar (after eliminating trivial non-terminals):

```

<element name='pdp'>
  <element name='rds' optional='true' type='time' units='ns' />
  <element name='med' optional='true' type='time' units='ns' />
  <element name='pp' optional='true' type='power' units='dBW' />
  <code>M=[</code>
  <repetition>
    <element name='ray'>
      <element name='time' type='time' units='ns' />
      <element name='power' type='power' units='dBW' />
    </element>
    <code>%time %power</code>
  </repetition>
  <code>];</code>
</element>

```

$(S_1) \quad S \rightarrow s(pdp), R, M, P, \{M=[\}, C, \{ \} i\}, e(pdp)$

$(R_1) \quad R \rightarrow \epsilon$

$(R_2) \quad R \rightarrow s(rds), d, e(rds)$

$(M_1) \quad M \rightarrow \epsilon$

$(M_2) \quad M \rightarrow s(med), d, e(med)$

$(P_1) \quad P \rightarrow \epsilon$

$(P_2) \quad P \rightarrow s(pp), d, e(pp)$

$(C_1) \quad C \rightarrow \epsilon$

$(C_2) \quad C \rightarrow \{B\}, s(ray), s(time), d, e(time),$
 $s(power), d, e(power), e(ray),$
 $\{\%time \%power\}, \{A\}, X, \{E\}$

$(X_1) \quad X \rightarrow \epsilon$

$(X_2) \quad X \rightarrow s(ray), s(time), d, e(time),$
 $s(power), d, e(power), e(ray),$
 $\{\%time \%power\}, \{A\}, X$

	$s(pdp)$	$s(rds)$	$s(med)$	$s(pp)$	$s(ray)$	$e(pdp)$
S	S_1					
R		R_2	R_1	R_1	R_1	R_1
M			M_2	M_1	M_1	M_1
P				P_2	P_1	P_1
C					C_2	C_1
X					X_2	X_1

Figure 3.7: (top) Binding schema for a power delay profile. rds , med , and pp stand for various optional statistics: rms delay spread, mean excess delay, and peak power. These statistics are ignored in this example. (left) L-attributed definition for a power delay profile. $\{B\}$, $\{A\}$, and $\{E\}$ stand for codes generated by the parser generator to handle repetitions. Otherwise, the meaning of $\{c\}$ is to print string c , followed by a new line character, after expanding element references. For clarity, full suffix factoring was not performed, but trivial productions were eliminated. (right) Predictive parsing table for a power delay profile.

```

<pdp>
  <rds>23.0998</rds>
  <med>20.5691</med>
  <pp>-75.5665</pp>
  <ray><time>-4</time><power>-88.0937</power></ray>
  <ray><time>-3</time><power>-82.4416</power></ray>
  <ray><time>-2</time><power>-78.5346</power></ray>
  <ray><time>-1</time><power>-76.2634</power></ray>
  <ray><time>0</time><power>-75.5665</power></ray>
  <ray><time>1</time><power>-76.4908</power></ray>
  <ray><time>2</time><power>-79.2101</power></ray>
  <ray><time>3</time><power>-84.0673</power></ray>
  <ray><time>24</time><power>-86.4976</power></ray>
  <ray><time>25</time><power>-84.3451</power></ray>
  <ray><time>26</time><power>-84.3173</power></ray>
  <ray><time>27</time><power>-85.963</power></ray>
  <ray><time>28</time><power>-87.7374</power></ray>
  <ray><time>29</time><power>-88.6525</power></ray>
  <ray><time>43</time><power>-89.2007</power></ray>
  <ray><time>44</time><power>-83.17</power></ray>
  <ray><time>45</time><power>-79.2179</power></ray>
  <ray><time>46</time><power>-77.3306</power></ray>
  <ray><time>47</time><power>-77.4917</power></ray>
  <ray><time>48</time><power>-79.645</power></ray>
  <ray><time>49</time><power>-83.6205</power></ray>
  <ray><time>50</time><power>-88.7676</power></ray>
</pdp>

```

M= [
-4 -88.0937
-3 -82.4416
-2 -78.5346
-1 -76.2634
0 -75.5665
1 -76.4908
2 -79.2101
3 -84.0673
24 -86.4976
25 -84.3451
26 -84.3173
27 -85.963
28 -87.7374
29 -88.6525
43 -89.2007
44 -83.17
45 -79.2179
46 -77.3306
47 -77.4917
48 -79.645
49 -83.6205
50 -88.7676
];

Figure 3.8: (left) An example PDP in XML. The data corresponds to a simulated channel in the corridor of the fourth floor of Durham Hall, Virginia Tech. The post processor samples the channel at 1 ns time intervals to match the output of a channel sounder. (right) Matlab encoding of the PDP on the left, output by the parser generated from the schema in Figure 3.7.

```

<selection id='s'> <sequence>
  <!-- empty -->
</sequence> <sequence>
  <code>c</code> <ref id='s' />
  <element name='x'> <code>b</code> </element>
</sequence> </selection>

```

$$S \rightarrow \epsilon$$

$$S \rightarrow \{c\}, S, s(x), \{b\}, e(x)$$

This grammar permits a derivation of the form $S \Rightarrow^+ \{c\}^k, (s(x), \{b\}, e(x))^k, k > 0$. However, code b cannot be executed before k is known since k executions of code c must precede the first execution of code b . Therefore, no LL(1) parser with stream semantics can parse documents that conform to this schema. On the other hand, removing $\{c\}$ from the L-attributed definition yields a grammar that is easily converted to LL(1):

$$S \rightarrow \epsilon \qquad S \rightarrow \epsilon$$

$$S \rightarrow S, s(x), \{b\}, e(x) \quad , \quad S \rightarrow s(x), \{b\}, e(x), S$$

This example is easy to generalize. □

Observation 2. Consider a set of all productions for a non-terminal A . Since any sequence $\{c_1\}\{c_2\}$ can be rewritten as $\{c\}$, where $c = c_1c_2$, we can uniquely represent this set by

$$A \rightarrow \{c_1\}A\alpha_1|\{c_2\}A\alpha_2|\cdots|\{c_n\}A\alpha_n|\beta_1|\beta_2|\cdots|\beta_m,$$

where no $\beta_j, 1 \leq j \leq m$, has a prefix $\{d\}A$. Immediate left recursion can be eliminated from this production without delaying user code execution if and only if

1. $c_1 = c_2 = \cdots = c_n = \epsilon$ (no user code to the left) or
2. $\left([(\beta_j \Rightarrow^* \gamma\{d\}\theta, 1 \leq j \leq m) \text{ or } (\alpha_i \Rightarrow^* \gamma\{d\}\theta, 1 \leq i \leq n)] \text{ implies } (d = \epsilon) \right)$ (no user code to the right) and $(c_1 = c_2 = \cdots = c_n)$ (same user code to the left).

In all other cases, execution of user code must be delayed until the last α_i is matched. □

Consider a derivation of A that is no longer left-recursive (i.e., does not have a prefix of $\{d\}A$). All such derivations can be written as

$$A \Rightarrow^+ \{c_{i_1}\}, \{c_{i_2}\}, \dots, \{c_{i_k}\}, \beta_j, \alpha_{i_k}, \dots, \alpha_{i_2}, \alpha_{i_1},$$

where $\beta_j, 1 \leq j \leq m$, stops left recursion after (at least) $k + 1$ steps and $1 \leq i_1, i_2, \dots, i_k \leq n$ represent the choices for α_i in the derivation. Suppose $\beta_j \Rightarrow^* \gamma\{d\}\theta$ or $\alpha_i \Rightarrow^* \gamma\{d\}\theta$. The sequence of codes $c_{i_1}, c_{i_2}, \dots, c_{i_k}$ must be executed before code d , but the LL(1) parser will only determine this sequence after it has parsed all of $\beta_j, \alpha_{i_k}, \dots, \alpha_{i_2}, \alpha_{i_1}$. Thus, eliminating left recursion entails delaying user code execution in all but the trivial cases mentioned above.

Example 2. Left factoring of L-attributed definitions poses similar problems. Consider the following schema and L-attributed definition (a more realistic version of this example would have a repetition in place of the x element):

```

<selection> <sequence>
  <code>c</code>
  <element name='x' /><element name='y' />
</sequence> <sequence>
  <code>d</code>
  <element name='x' /><element name='z' />
</sequence> </selection>

```

$$S \rightarrow \{c\}, s(x), e(x), s(y), e(y)$$

$$S \rightarrow \{d\}, s(x), e(x), s(z), e(z)$$

The decision about whether to execute code c or d cannot be made until $s(y)$ or $s(z)$ is processed. However, removing user codes makes this L-attributed definition easy to refactor. Again, we can show a more general condition. \square

Observation 3. Consider a set of all productions for a non-terminal A written as

$$A \rightarrow \alpha_1\beta_1|\alpha_2\beta_2|\cdots|\alpha_n\beta_n|\gamma_1|\gamma_2|\cdots|\gamma_m,$$

such that $\alpha'_1 = \alpha'_2 = \cdots = \alpha'_n = \alpha \neq \epsilon$ (α' denotes α with all user code removed) and α is not a prefix of any $\gamma'_1, \gamma'_2, \dots, \gamma'_m$. Let the length of α be maximum and the lengths of $\alpha_i, 1 \leq i \leq n$, be minimum subject to $n \geq 2$, in which case this representation of A is unique. A can be left-factored without delaying execution of user code if and only if

1. no rewriting of A in the above form exists (no two definitions of A share the same prefix, less user codes), or
2. $\alpha_1 = \alpha_2 = \cdots = \alpha_n$ (same codes to the left) and $A \rightarrow \gamma_1|\gamma_2|\cdots|\gamma_m$ can be left-factored. \square

To summarize, we implement bindings in terms of L-attributed definitions from parsing theory. These bindings work well in practice, but, in theory, annotating a schema that can be rewritten in LL(1) form can make it no longer rewritable in LL(1) form. This difficulty is inherent in L-attributed definitions. We currently assume that the user is responsible for resolving such conflicts. In practice, schemas for PSE data rarely require complicated grammars. Repetitions take care of most of the recursive schema definitions. To make LL(1) parsing possible, troublesome content can be simply enclosed in an extra XML element, whose start and end tags disambiguate the transitions of the LL(1) parser.

3.5 Conversion

Conversion is the cornerstone of a system's ability to handle changes and interface mismatches. Conversion in a PSE helps to retain historical data and facilitates inclusion of new components. We use change detection principles from [17], with a few important differences. First, our goal is not merely to detect changes, but to make PSE components work despite the changes. Second, we detect changes in the schema, not in the data. The PSE environment must guarantee that the data is in the right format for the component. The job of the component is to process any data instance that conforms to the right format. Last, change detection and conversion are local to the extent possible. Locality is a virtue not only because it allows for stream processing, but also because it limits sporadic conversions between unrelated entities.

Similarly to the two previous sections, this section starts with a comprehensive example. Then, we describe the core of the conversion algorithm and outline its limitations. Finally, we extend the initial algorithm to handle content replacements: unit conversion and user-defined conversion filters. At this point, it should not come as a surprise to the reader that most of the technical limitations of conversion are due to binding codes, not to the nature of the schema language. Therefore, the tedious details of handling binding codes are omitted. The emphasis is on non-technical limitations. What forms of semantic conversions can be 'syntactized' in a schema language? When does such 'syntactization' back fire and produce undesired outcomes?

The functional statement of the conversion problem can be given as follows. Given the actual schema S_a and the required schema S_r , replace binding codes in S_a with binding codes in S_r and conversion codes to obtain the conversion schema S_c . S_c must describe precisely the documents described by S_a , but perform the same bindings as S_r .

Example 3. Figure 3.9 depicts two slightly different schemas for antenna descriptions in S⁴W. The schema at the bottom (actual schema) was our first attempt at defining a data format for antenna descriptions. This version supported only one antenna type and exhibited several inadequate representation choices. E.g., polar coordinates should have been used instead of Cartesian coordinates because antenna designers prefer to work in the polar coordinate system. Antenna gain was not considered in the first version because its effect is the same as changing transmitter power. However, this seemingly unnecessary parameter should have been included because it results in a more direct correspondence of simulation input to a physical system.

The schema at the top of Fig. 3.9 (required schema) improves upon the actual schema in several ways. It better adheres to common practices and supports more antenna types. However, this schema is different from the actual schema, while compatibility with old data needs to be retained (requirement 2). Figure 3.10 illustrates how addition of conversion and binding codes to the actual schema solves the compatibility problem. A parser generated from the conversion schema in Figure 3.10 will recognize the actual data and provide the required binding. □

Following [17], the basic assumption of the conversion algorithm is that the actual schema S_a can be converted to the required schema S_r by some sequence of ‘standard’ edits. This sequence of edits is called an *edit script*. Once the possible types of edits are defined (what we can call a ‘conversion library’), the job of the conversion algorithm is to (a) find an edit script that transforms the actual schema S_a to the required schema S_r and (b) express this edit script as data transformations, not schema transformations. In other words, the conversion algorithm looks for a systematic procedure that converts actual data instances that conform to S_a to the required format S_r . This procedure is expressed as a conversion schema S_c that has the structure of S_a , but binding codes from S_r and the conversion library. S_c is then used to generate a parser that parses data instances conforming to S_a and acts as if it parsed data instances conforming to S_r .

Our conversion algorithm supports four kinds of schema edits:

1. generalization,
2. restriction,
3. reordering, and
4. replacement.

We use these terms in reference to the required schema, e.g., ‘the required schema is a generalization of the actual schema.’ Generalization and restriction of schema trees are similar to insertions and deletions in sequence alignment problems. Reordering and replacement mostly retain their standard meaning, except we consider replacements of sets of schema blocks, not individual schema blocks. We first reduce the problem of converting trees to an easier problem of converting sequences (see Figure 3.11). Sequence conversion (rule Q) in this initial formulation performs all conversions but replacements. Then, we slightly restrict this definition to make it practical and generalize rule Q to accommodate replacements (unit conversion and user-defined conversion filters).

The conversion algorithm revolves around the ‘determines’ relation between schemas. Intuitively, an actual schema S_a should determine a required schema S_r if any document that conforms to S_a contains sufficient information to construct an ‘appropriate’ document that conforms to S_r . ‘Appropriate’ here is obviously a domain-specific notion, and in the absence of a domain theory, there is no hard and fast measure of ‘appropriateness.’ Given two slightly different schemas, only a domain expert can tell whether or not it is meaningful to attempt a conversion from one form to another. Therefore, our conversion rules should be viewed as heuristics that we have found to be useful enough to be supported in a conversion library. They are neither sound nor complete in an algorithmic sense (because we do not have an objective, external, measure

```

<element name='antennas'>
  <repetition>
    <element name='antenna'>
      <element name='id' type='string' min='1' />
      <element name='phi' type='angle' />
      <element name='theta' type='angle' />
      <element name='gain' type='ratio' units='dB' optional='true' default='0' />
      <code>puts stdout "%id: %phi %theta %gain"</code>
      <selection>
        <element name='waveguide'>
          <element name='width' type='distance' units='mm' />
          <element name='height' type='distance' units='mm' />
          <code>puts stdout "waveguide: %width %height"</code>
        </element>
        <element name='pyramidal_horn'>
          <element name='width' type='distance' units='mm' />
          <element name='rw' type='distance' units='mm' />
          <element name='height' type='distance' units='mm' />
          <element name='rh' type='distance' units='mm' />
          <code>puts stdout "pyramidal horn: %width %rw %height %rh"</code>
        </element>
      </selection>
    </element>
  </repetition>
</element>

<element name='antennas'>
  <repetition>
    <element name='antenna'>
      <element name='id' type='string' min='1' />
      <element name='description' type='*' />
      <element name='x' type='coordinate' />
      <element name='y' type='coordinate' />
      <element name='z' type='coordinate' />
      <element name='waveguide'>
        <element name='width' type='distance' units='in' />
        <element name='height' type='distance' units='in' />
      </element>
    </element>
  </repetition>
</element>

```

Figure 3.9: Two slightly different schemas for a collection of antennas. The component requires the top schema, but the data conforms to the bottom schema. The bottom schema (a) represents antenna orientation in Cartesian coordinates, not polar coordinates, (b) lacks antenna gain, (c) requires antenna descriptions, (d) measures antenna dimensions in inches, not millimeters, and (e) covers only one antenna type. The schema at the bottom does not contain binding codes because they are irrelevant for this example. All binding codes are in Tcl.

```

<element name='antennas'>
  <repetition>
    <element name='antenna'>
      <element name='id' type='string' min='1' />
      <element name='description' type='*' />
      <element name='x' type='coordinate' />
      <element name='y' type='coordinate' />
      <element name='z' type='coordinate' />
      <code> <!-- convert coordinates from rectangular to polar -->
        set _r [expr sqrt(%x*%x+%y*%y+%z*%z)]
        set %phi [expr atan2(%y,%x)]
        set %theta [expr acos(%z/$_r)]
      </code>
      <code> <!-- set default gain -->
        set %gain 0
      </code>
      <code>puts stdout "%id: %phi %theta %gain"</code>
    <element name='waveguide'>
      <element name='width' type='distance' units='mm' />
      <code> <!-- convert units from inches to millimeters -->
        set %width [expr 25.4*%width]
      </code>
      <element name='height' type='distance' units='mm' />
      <code> <!-- convert units from inches to millimeters -->
        set %height [expr 25.4*%height]
      </code>
      <code>puts stdout "waveguide: %width %height"</code>
    </element>
  </element>
</repetition>
</element>

```

Figure 3.10: Actual schema from Figure 3.9 (bottom) after inserting conversion and binding codes. This schema describes the actual documents, but provides the bindings of the required schema (top of Figure 3.9). We use `_r` instead of `%r` because the latter could interfere with another use of the name `r`.

D_r	$\text{data}(\text{base}_a, \text{min}_a, \text{max}_a, \text{number}_a, \text{finite}_a, \text{units}_a) \succeq$ $\text{data}(\text{base}_r, \text{min}_r, \text{max}_r, \text{number}_r, \text{finite}_r, \text{units}_r)$ if $\text{base}_a = \text{base}_r, \text{min}_a \geq \text{min}_r, \text{max}_a \leq \text{max}_r, \text{number}_r \Rightarrow \text{number}_a, \text{finite}_r \Rightarrow \text{finite}_a,$ $\text{units}_a = \text{units}_r$
E	$\text{element}(\text{id}_a, \text{opt}_a, \text{name}_a, C_{a1}, C_{a2}, \dots, C_{an}) \succeq \text{element}(\text{id}_r, \text{opt}_r, \text{name}_r, C_{r1}, C_{r2}, \dots, C_{rm})$ if $\text{name}_a = \text{name}_r, \text{opt}_a \Rightarrow \text{opt}_r, Q_a(C_{a1}, C_{a2}, \dots, C_{an}) \succeq Q_r(C_{r1}, C_{r2}, \dots, C_{rm})$
E_g	$X_a(\text{id}_a, \text{opt}_a, \dots) \succeq \text{element}(\text{id}_r, \text{opt}_r, \text{name}_r, C_{r1}, C_{r2}, \dots, C_{rm})$ if $\text{opt}_a \Rightarrow \text{opt}_r, Q_a(X_a(\text{id}_a, \text{opt}_a, \dots)) \succeq Q_r(C_{r1}, C_{r2}, \dots, C_{rm})$
E_r	$\text{element}(\text{id}_a, \text{opt}_a, \text{name}_a, C_{a1}, C_{a2}, \dots, C_{an}) \succeq X_r(\text{id}_r, \text{opt}_r, \dots)$ if $\text{opt}_a \Rightarrow \text{opt}_r, Q_a(C_{a1}, C_{a2}, \dots, C_{an}) \succeq X_r(\text{id}_r, \text{opt}_r, \dots)$
P	$\text{sequence}(\text{id}_a, \text{opt}_a, C_{a1}, C_{a2}, \dots, C_{an}) \succeq \text{sequence}(\text{id}_r, \text{opt}_r, C_{r1}, C_{r2}, \dots, C_{rm})$ if $\text{opt}_a \Rightarrow \text{opt}_r, Q_a(C_{a1}, C_{a2}, \dots, C_{an}) \succeq Q_r(C_{r1}, C_{r2}, \dots, C_{rm})$
P_g	$X_a(\text{id}_a, \text{opt}_a, \dots) \succeq \text{sequence}(\text{id}_r, \text{opt}_r, C_{r1}, C_{r2}, \dots, C_{rm})$ if $\text{opt}_a \Rightarrow \text{opt}_r, Q_a(X_a(\text{id}_a, \text{opt}_a, \dots)) \succeq Q_r(C_{r1}, C_{r2}, \dots, C_{rm})$
P_r	$\text{sequence}(\text{id}_a, \text{opt}_a, C_{a1}, C_{a2}, \dots, C_{an}) \succeq X_r(\text{id}_r, \text{opt}_r, \dots)$ if $\text{opt}_a \Rightarrow \text{opt}_r, Q_a(C_{a1}, C_{a2}, \dots, C_{an}) \succeq X_r(\text{id}_r, \text{opt}_r, \dots)$
C	$\text{selection}(\text{id}_a, \text{opt}_a, C_{a1}, C_{a2}, \dots, C_{an}) \succeq \text{selection}(\text{id}_r, \text{opt}_r, C_{r1}, C_{r2}, \dots, C_{rm})$ if $\text{opt}_a \Rightarrow \text{opt}_r, \forall C_{ai} : (\exists! C_{rj} : C_{ai} \succeq C_{rj})$
C_g	$X_a(\text{id}_a, \text{opt}_a, \dots) \succeq \text{selection}(\text{id}_r, \text{opt}_r, C_{r1}, C_{r2}, \dots, C_{rm})$ if $\text{opt}_a \Rightarrow \text{opt}_r, (\exists! C_{rj} : X_a(\text{id}_a, \text{opt}_a, \dots) \succeq C_{rj})$
R	$\text{repetition}(\text{id}_a, \text{opt}_a, \text{min}_a, \text{max}_a, C_{a1}, C_{a2}, \dots, C_{an}) \succeq$ $\text{repetition}(\text{id}_r, \text{opt}_r, \text{min}_r, \text{max}_r, C_{r1}, C_{r2}, \dots, C_{rm})$ if $\text{min}_a \geq \text{min}_r, \text{max}_a \leq \text{max}_r, \text{opt}_a \Rightarrow \text{opt}_r, Q_a(C_{a1}, C_{a2}, \dots, C_{an}) \succeq Q_r(C_{r1}, C_{r2}, \dots, C_{rm})$
R_g	$X_a(\text{id}_a, \text{opt}_a, \dots) \succeq \text{repetition}(\text{id}_r, \text{opt}_r, \text{min}_r, \text{max}_r, C_{r1}, C_{r2}, \dots, C_{rm})$ if $\text{min}_r \leq 1, \text{max}_r \geq 1, \text{opt}_a \Rightarrow \text{opt}_r, Q_a(X_a(\text{id}_a, \text{opt}_a, \dots)) \succeq Q_r(C_{r1}, C_{r2}, \dots, C_{rm})$
F	$\text{ref}(\text{id}_a) \succeq \text{ref}(\text{id}_r)$ if $X_a(\text{id}_a, \text{opt}_a, \dots) \succeq X_r(\text{id}_r, \text{opt}_r, \dots)$
Q	$Q_a(C_{a1}, C_{a2}, \dots, C_{an}) \succeq Q_r(C_{r1}, C_{r2}, \dots, C_{rm})$ if $\forall C_{rj}(\dots, \text{opt}_{rj}, \dots) : [(\exists! C_{ai} : C_{ai} \succeq C_{rj}) \text{ or } (\text{opt}_{rj})]$

Figure 3.11: Version 1 of the ‘determines’ relation $X_a(\text{id}_a, \text{opt}_a, \dots) \succeq X_r(\text{id}_r, \text{opt}_r, \dots)$ between an actual schema block $X_a(\text{id}_a, \text{opt}_a, \dots)$ and a required schema block $X_r(\text{id}_r, \text{opt}_r, \dots)$. We use the non-XML notation from Figure 3.4 plus $X_a(\text{id}_a, \text{opt}_a, \dots)$ and $X_r(\text{id}_r, \text{opt}_r, \dots)$ are shortcuts for any schema block (data blocks are never optional and have empty ids). \Rightarrow means logical implication and $\exists!$ means ‘there exists a unique.’ The rules are applied top to bottom, left to right. The first matching rule wins (no backtracking). This definition will be later restricted to make it computable and rule Q will be extended to handle replacements.

of ‘conversion correctness’). Instead, they represent a tradeoff between soundness and completeness and should be carefully evaluated for use in a particular domain. With this disclaimer in mind, version 1 of the determines relation between S_a and S_r (S_a determines S_r ; $S_a \succeq S_r$) is defined in Figure 3.11. We will also find the notion of schema equivalence useful: we say that two schemas S_a and S_r are *equivalent* if $S_a \succeq S_r$ and $S_r \succeq S_a$.

The first rule (D_r) in Figure 3.11, for instance, says that a value of primitive type (‘data’) can be substituted for another if they have the same base type, their ranges are compatible, and they have the same units. It ensures that all primitive type constraints of S_r are met by S_a (restriction). Thus, D_r is simply a definition of type derivation by range restriction (the ‘r’ subscript in this and other rules stands for restriction; similarly, the ‘g’ subscript stands for generalization). Rules E , P , and R state the obvious: two black boxes are compatible if they have compatible wrappers (restriction) and compatible contents (any conversions). Rule C says that any choice in S_a must uniquely determine some choice in S_r (restriction). Rule Q enforces that every block in S_r is uniquely determined by some block in S_a . This formulation of rule Q ignores extra blocks in S_a (restriction), permits optional elements in S_r to be unmatched (generalization), and allows for contents reordering. Rule F deals with references. Only rules D_r , E , P , C , and R are sound. Rule F looks sound, but it makes the determines relation not computable. Rule Q is unsound primarily because it ignores ‘unnecessary’ blocks in S_a .

Rules E_g , P_g , C_g , and R_g handle generalizations across schema blocks of (possibly) different types. Their counterparts E_r and P_r handle symmetric restrictions (why is there no C_r or R_r ?). Rule C_g was demonstrated in the example above. It is a base case for rule C . Rule C_g states that one way to generalize a schema block is to enclose it in a selection, i.e., provide more choices in S_r than were available in S_a . This rule is sound. Rules E_g , P_g , and R_g have similar motivations, but they are unsound. Essentially, we assume that decorating any black box with any number of wrappers does not change the meaning of the black box (generalization). Similarly, we assume that wrappers can be freely removed to expose the black box (restriction).

Consider a sequence of schemas that describes some physical system in progressively greater detail. Suppose some subsystem is described by a single parameter. Common practice is to allocate a single schema block to this subsystem. What happens when a more detailed description of this subsystem is incorporated into the schema? Chances are, the original schema block allocated to the subsystem will be either (a) augmented with more contents (restriction part of rule Q) or (b) wrapped in another block. The generalization and restriction rules handle case (b). However, blind application of these rules can lead to disaster because these rules disregard some semantic information. Examples will make these points clearer.

Example 4. One common trick used to improve wireless system performance is space-time transmit diversity (STTD). Instead of a single transmitter antenna, the base station uses two transmitter antennas separated by a small distance. PDPs are very sensitive to device positioning, so two uncorrelated transmitter antennas can produce widely different signals at the same receiver location. If the signal from one of the antennas is weak, the signal from another antenna will probably be strong, so the overall performance is expected to improve. Consider how addition of STTD to the ray tracer affects the schema of the transmitter file. The original schema is on the left and the new schema (with STTD support) is on the right. The second antenna is optional because STTD is not used in every system due to cost considerations.

```

<element name='tx'>
  <ref id='coordinates' />
  <element name='power' type='power' />
  <element name='freq' type='double' />
</element>
<element name='base_station'>
  <element name='tx'>
    <ref id='coordinates' />
    <element name='power' type='power' />
    <element name='freq' type='double' />
  </element>
  <element name='tx' optional='true'>
    <ref id='coordinates' />
    <element name='power' type='power' />
    <element name='freq' type='double' />
  </element>
</element>

```

The new ray tracer should be able to work with old data because it supports one or two transmitter antennas. The old ray tracer should be able to work with new data, albeit the results will be approximate when the new data contains two transmitter antennas. Further generalizing this example to n transmitter antennas would require a repetition. We support conversion to repetitions, but not from repetitions. For this example, we could extract any antenna because they usually have the same parameters and are positioned close together. However, we cannot extract an arbitrary ray from a PDP because the ray with maximum power is usually intended. Extracting any other ray would typically produce nonsense results. \square

Example 5. Havoc can result if rules E_r and E_g are applied to the same element. Element names have semantic meaning, but this particular composition of rules allows arbitrary renaming of elements. Such renaming would make the following two schemas equivalent.

```

<element name='tx_gain' type='ratio' />
  <element name='snr' type='ratio' />

```

Even though both transmitter antenna gain and signal-to-noise ratio are ratios measured in the same units (dB), they convey largely different information. We avoid such blatant mistakes by limiting the application of generalization and restriction rules. In particular, no element can be renamed. \square

As the last example illustrates, the ‘determines’ relation in Figure 3.11 needs to be restricted. It is helpful to redefine this relation in terms of a context-free grammar that describes $S_a S_r$. Let the terminals be `element()`, `sequence()`, `selection()`, `repetition()`, `ref()`, `data()`, and all element names and other values used in two schemas under consideration. Let the non-terminals be the labels of the rules in Figure 3.11, a special start non-terminal A , and intermediate non-terminals introduced by the rules. We can formally define the necessary restrictions by limiting the shape of the parse tree for $S_a S_r$. Consider a path $R_1, R_2, \dots, R_n, n > 0$, from some internal node $R_1 \neq A$ to some internal node $R_n \neq A$, where all $R_i, 1 \leq i \leq n$, are rule labels. If \mathcal{R} is the set of restriction rules and \mathcal{G} is the set of generalization rules, we require that $(R_i \in \mathcal{R})$ implies $(R_{i-1} \notin \mathcal{G} \text{ and } R_{i+1} \notin \mathcal{G})$, i.e., restriction and generalization rules cannot be applied in sequence. This restriction of the parse tree disallows renaming of elements, but does not limit the number of wrappers around black boxes. Bounded determination deals with the latter problem. We say that S_a k -determines S_r ($S_a \succeq^k S_r$) if no path R_1, R_2, \dots, R_n contains a substring of (possibly different) generalization (restriction) rules of length greater than k . We leave it up to the reader to appropriately restrict rule F (reference). These restrictions make the ‘determines’ relation computable and enforce locality of conversions. As a side effect, we have shown that the problem of constructing a conversion schema S_c from the actual schema S_a and the required schema S_r can be reduced to validation

and binding (parsing and translation). However, schema conversion need not work with streams of data, so a parser more powerful than a predictive parser should be used.

It remains to consider requirements 4 and 5: unit conversion and user-defined conversion filters (replacements). Let D be a set of all primitive types derived from double (recall that a primitive type is defined by the base type, the range of legal values, and a unit expression). Unit conversion, e.g., converting kg/m^2 to lb/in^2 , is the simpler of the two replacements. Both actual and required unit expressions are converted to a canonical form (e.g., a fraction of products of sums of CI units or dB) and then the conversion function is found. Unit conversions are functions of the form

$$U : D_a \rightarrow D_r,$$

where $D_a, D_r \in D$ are specific primitive types. User-defined conversion filters are functions of the form

$$H : D_{a1} \times D_{a2} \times \dots \times D_{an} \rightarrow D_{r1} \times D_{r2} \times \dots \times D_{rm},$$

where $n, m > 0$ and all $D_{ai}, D_{rj} \in D, 1 \leq i \leq n, 1 \leq j \leq m$, are specific primitive types. Arithmetic operators and common mathematical functions are allowed in user-defined conversion filters. Each user-defined conversion filter is tagged with element names $name_{ea1}, name_{ea2}, \dots, name_{ean}$ and $name_{er1}, name_{er2}, \dots, name_{erm}$ that determine when the filter applies. Such filters define rules of the form

$$\begin{aligned} &(\text{element}(\$ \$, name_{ea1}, D_{a1}), \text{element}(\$ \$, name_{ea2}, D_{a2}), \dots, \text{element}(\$ \$, name_{ean}, D_{an})) \succeq \\ &(\text{element}(\$ \$, name_{er1}, D_{r1}), \text{element}(\$ \$, name_{er2}, D_{r2}), \dots, \text{element}(\$ \$, name_{erm}, D_{rm})). \end{aligned}$$

Both kinds of filters are compiled into codes such as shown in Figure 3.10. Rule Q is modified to take advantage of replacements. Basically, we are looking for (unique) partitions of the actual schema blocks $C_{a1}, C_{a2}, \dots, C_{an}$ and required schema blocks $C_{r1}, C_{r2}, \dots, C_{rm}$ such that each set of schema blocks in the required partition is determined by some set of schema blocks in the actual partition. Determination can proceed through the rules in Figure 3.11, unit conversions, and user-defined conversion filters (if everything else fails, optional blocks in the required schema can remain unmatched).

The ultimate goal of the conversion algorithm is to find a meaningful edit script. However, this goal is impossible to achieve without knowledge of the domain. What happens when several edit scripts exist, i.e., the problem of finding an edit script is ambiguous? Depending on the nature of the ambiguity, we can choose any edit script, the minimal (in some sense) edit script, or to refuse to perform conversion. The conversion algorithm described here either settles for some local minimum (e.g., rule E is preferred over rule E_g) or requires uniqueness of conversions (rules C, C_g , and most of rule Q). Ambiguity remains an open problem that is unlikely to be solved by a syntactic conversion algorithm. Following the principle of least user astonishment, we choose to reject most of ambiguous conversions.

Finally, let us consider how binding codes limit conversion. We omit formal treatment of the problem and limit the discussion to an example. It is easy to see that conversion may require delaying binding code execution. This should not be surprising since one kind of conversion is reordering.

Example 6. Consider a required schema with binding codes (left) and an actual schema (right).

```

<sequence>
  <element name='a' type='double' />
  <code>c1</code>
  <repetition>
    <ref id='b' />
    <code>c2</code>
  </repetition>
</sequence>
  <sequence>
    <repetition><ref id='b' /></repetition>
    <element name='x' type='double' />
    <element name='y' type='double' />
  </sequence>

```

Assume that there exists a user-defined conversion filter that calculates a from x and y . If we ignore binding code c_2 , conversion is clearly local. However, conversion with c_2 present will require delaying all executions of c_2 until c_1 is executed. The latter can only happen when the last piece of the schema is matched. In other words, binding codes should be placed as late as possible in the schema. \square

This section presented a number of local conversions appropriate for PSE data. Conversions are carried out by extra codes injected in the actual schema. The conversion algorithm was built around the ‘determines’ relation between schemas. The algorithm has some technical limitations related to binding codes, but its major limitation is conceptual. Conversion, in the form presented here, is syntactic. It is based on the weak semistructured data model, not on the underlying domain theory (wireless communications). Therefore, we can only speculate about the causes of differences between the actual and required schemas. There is no guarantee that automatic conversion will produce meaningful results. A stronger data model is necessary to perform complex, yet meaningful, conversions.

3.6 Integration with a PSE

A complete PSE requires functionality far beyond validation, binding, and conversion. BSML ensures that the components can read streams of XML data, but it does not support tasks such as scheduling, communication, database storage and retrieval, connecting multiple components into a given topology, and computational steering. We broadly call software that performs all of these tasks an *execution manager*. Figure 3.12 illustrates how BSML software and the execution manager function together.

From a systems point of view, BSML schemas are metadata and the BSML software is a parser generator. Recall that the parser generator generates parsers that perform validation, binding, and conversion functions (every such generated parser will be able to take input data and stream it through the component). Both the data and the metadata are stored in a database. We can distinguish three kinds of metadata: schemas, component metadata, and model instance metadata. Only one form of metadata (schemas) was described in this chapter. Component metadata contains component’s local parameters, such as executable name, programming language, and input/output port schemas. It is the kind of metadata used in CCAT. Model instance metadata, i.e., component topology and other global execution parameters, serves a purpose similar to GALE’s workflow specifications. It supports our requirement 3.

A parser is lazily generated for each used combination of component’s input port schema (required schema) and the schema of the data instance connected to this port (actual schema). Component metadata specifies how linking must be performed (e.g., which of the three kinds of bindings to use). Component instances are further managed by the execution manager. Model instance metadata specifies how to execute the model instance (e.g., the topology and the number of processors), while model instance data serves as the actual (data) input to the model instance. To summarize, the BSML parser generator creates component instances—programs that take a number of XML streams as inputs and produce a number of XML streams as outputs. This representation is appropriate for management of a PSE execution environment.

3.6.1 Status of Prototype

In S⁴W, the execution manager is implemented in Tcl/Tk and most of the component metadata is hard-coded. Model instance metadata consists primarily of the number of processors and a cross-product of references to model instance data. An (incomplete) example of such a specification is

‘compute power coverage maps for these three transmitter locations in Torgersen Hall and show a graph of BERs with the signal-to-noise ratio varying from zero to twenty dB in steps of two dB; use thirty nodes of a 200-node Beowulf cluster.’

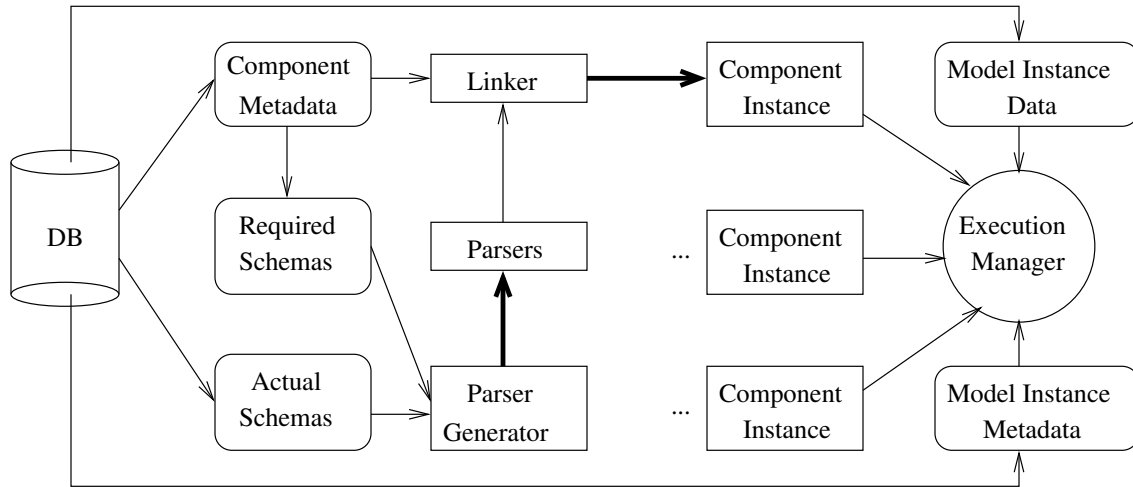


Figure 3.12: BSML integration with PSE execution environment. The BSML parser generator creates parsers that handle input ports of each component. Execution manager controls the execution of a model instance that consists of components, model instance data, and model instance metadata. Figure 3.1 partially defines one such instance.

PostgreSQL and the filesystem serve the role of the database. Large files (e.g., floor plans) are typically stored in the filesystem and small ones (e.g., PDPs) are usually imported into PostgreSQL. The parser generator is written in SWI Prolog. It generates parsers in Tcl. This choice of languages was driven by the expertise of the people involved in the project. It is not a result of any systematic investigation. Currently, the generated parsers are used mostly in the execution manager, visualization components, and database interfacing components.

3.7 Discussion of BSML

We have described the use of validation, binding, and conversion facilities to solve data interchange problems in a PSE. Since all three concepts are closely related to parsing and translation, viewing application composition in terms of data management uncovers well-understood solutions to interface mismatch problems. The semistructured data model allows us to syntactically define several forms of conversions that are usually implemented by hand-written mediators in PSEs. Such automation reduces the cost of PSE development and, more importantly, brings PSEs closer to their ultimate goal—namely, PSE users should be solving their domain-specific problems, not be beset by the technical details of component composition in a heterogeneous computing environment.

Several extensions to the present work are envisioned. First, the expressiveness of schema languages for data interchange and application composition can be formally characterized. This will allow us to reason about requirements such as stream processing from a modeling perspective. Such a study will also lead to a better understanding of the roles that a markup language can play in a PSE. Second, dataflow relationships between components can be made explicit. BSML guarantees that any component instance be able to process streams of data, but synchronization issues are meant to be resolved by the execution manager. Tighter integration of BSML and composition frameworks can be explored. Finally, the overall view of a PSE as a semistructured data management system deserves further exploration. For example, it seems possible

to automatically generate workflow specifications from queries on a semistructured database of simulation results.

Chapter 4

Using Hierarchical Data Mining to Characterize Performance of Wireless System Configurations

This chapter presents a statistical framework for assessing wireless systems performance using hierarchical data mining techniques. We consider WCDMA (wideband code division multiple access) systems with two-branch STTD (space time transmit diversity) and 1/2 rate convolutional coding (forward error correction codes). Monte Carlo simulation estimates the bit error probability (BEP) of the system across a wide range of signal-to-noise ratios (SNRs). A performance database of simulation runs is collected over a targeted space of system configurations. This database is then mined to obtain regions of the configuration space that exhibit acceptable average performance. The shape of the mined regions illustrates the joint influence of configuration parameters on system performance. The role of data mining in this application is to provide explainable and statistically valid design conclusions. The research issue is to define statistically meaningful aggregation of data in a manner that permits efficient and effective data mining algorithms. We achieve a good compromise between these goals and help establish the applicability of data mining for characterizing wireless systems performance.

Data mining is becoming increasingly relevant in simulation methodology and computational science [43]. Data mining entails the ‘non-trivial process of identifying valid, novel, potentially useful, and ultimately understandable patterns in data’ [23]. It can be used in both predictive (e.g., quantitative assessment of factors on some performance metric) and descriptive (e.g., summarization and system characterization) settings. Our goal in this chapter is to demonstrate a hierarchical data mining framework applied to the problem of characterizing wireless system performance.

We study the effect of configuration parameters on the bit error probability (BEP) of a system simulated in S^4W . The approach we take is to accumulate a performance database of simulation runs that sweep over a targeted space of system configurations. This database is then mined to obtain regions of the configuration space that exhibit acceptable average performance. Exploiting prior knowledge about the underlying simulation, organizing the computational steps in data mining, and interpreting the results at every stage, are important research issues. In addition, we bring out the often prevailing tension between making statistically meaningful conclusions and the assumptions required for efficient and effective data mining algorithms. This interplay leads to a novel set of problems that we address in the context of the wireless systems performance domain.

Data mining algorithms work in a variety of ways but, for the purposes of this chapter, it is helpful to think of them as performing systematic aggregation and redescription of data into higher-level objects. Our

work can be viewed as employing three such layers of aggregation: points, buckets, and regions. Points (configurations) are records in the performance database. These records contain configuration parameters as well as unbiased estimates of bit error probabilities that we use as performance metrics. Buckets represent averages of points. We use buckets to reduce data dimensionality to two, which is the most convenient number of dimensions for visualization. Finally, buckets are aggregated into 2D regions of constrained shape. We find regions of buckets where we are most confident that the configurations exhibit acceptable average performance. The shapes of these regions illustrate the nature of the joint influence of the two selected configuration parameters on the configuration performance. Specific region attributes, such as region width, provide estimates for the thresholds of sensitivity of configurations to imbalance in parameter values.

4.1 In this Chapter

Our major contribution is the development of a statistical framework for assessing wireless system performance using data mining techniques. The following section outlines wireless systems performance simulation methodology and develops a statistical framework for spatial aggregation of simulation results. Section 4.3 demonstrates a substantial subset of this framework in the context of a performance study of WCDMA (wideband code division multiple access [35]) systems that employ two-branch STTD (space-time transmit diversity [5]) techniques and 1/2 rate convolutional coding (forward error correction codes [35]). We study the effect of power imbalance between the branches on the BEP of the system across a wide range of average signal-to-noise ratios (SNRs). Section 4.4 extends the statistical framework to support computation of optimized regions of the bucket space. Such regions are computed by a well-known data mining algorithm [27, 52]. Section 4.5 applies these concepts to the example in Section 4.3. Section 4.6 summarizes present findings and outlines directions for future research.

4.2 The Statistics of Aggregation and the Aggregation of Statistics

Temporal variations in wireless channels have been extensively studied in the literature [53]. The present work uses a Monte Carlo simulation of WCDMA wireless systems to study the effect of these variations. The simulation traces a number of frames of random information bits through the encoding filters, the channel (a Rayleigh fading linear filter [32]), and the decoding filters. The inputs are hardware parameters, average SNR, channel impulse response, and the number of frames to simulate. The output is the bit error rate—the ratio of the number of information bits decoded in error to the total number of information bits simulated. Simulations of this kind statistically model channel variations due to changes in the environment and device movement across a small geographical area (*small-scale fading* [32]). We refer to this kind of channel variation as *temporal variation* because a system is simulated over a period of time. Further, we say that a given list of inputs to the WCDMA simulation is a *configuration* or a *point* in the configuration space.

Spatial variations are due to changes in system configurations. We use this term to describe two quite different phenomena: changes in the average SNR and channel impulse response due to *large-scale fading* [32] and variations of hardware parameters. A typical approach to the analysis of spatial variations is to run several temporal variation simulations (i.e., compute BERs at several points within a given area of interest) and plot 1D or 2D slices of the configuration space, as shown in Figure 4.1. In this chapter, we augment this approach with statistically meaningful aggregation of performance estimates across several points. The result of this aggregation is a space of buckets, each bucket representing the aggregation of a number of points. Moving up one level of aggregation in this manner allows us to bring data mining algorithms to

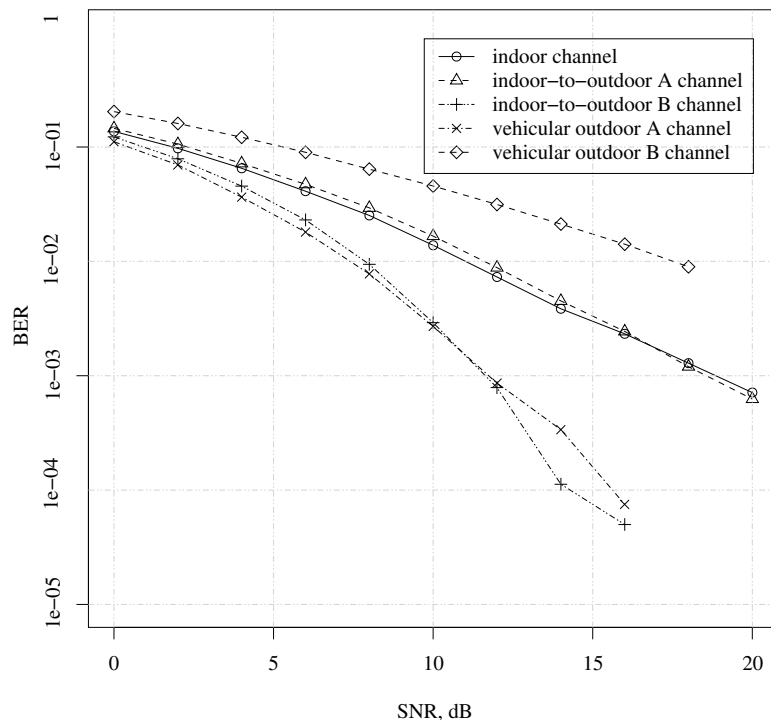


Figure 4.1: Typical 1D slices of the configuration space. The plots show simulated BERs (bit error rates) of wireless systems for five common benchmark channels [49] across a typical range of average SNRs.

c, C, R	entities (points, buckets, regions)
x, b, B	random variables
$E[x], E[b], E[B]$	true means of random variables x, b, B
σ^2, Σ^2	true variances of random variables b, B
$\hat{x}, \hat{b}, \hat{B}$	estimates of means $E[x], E[b], E[B]$ of random variables x, b, B
$\hat{\sigma}^2, \hat{\Sigma}^2$	estimates of variances σ^2, Σ^2 of random variables b, B
$P(E)$	probability of event E , where E is a boolean condition
$F_{N-1}(T)$	$P(X < T)$ for X having the Student t distribution with $N - 1$ degrees of freedom
$\{x_k\}_{k=1}^n$	set $\{x_1, x_2, \dots, x_n\}$
$\{\{x_{kj}\}_{j=1}^{n_k}\}_{k=1}^n$	set $\{x_{11}, x_{12}, \dots, x_{1n_1}, x_{21}, x_{22}, \dots, x_{2n_2}, \dots, x_{n1}, x_{n2}, \dots, x_{nn_n}\}$

Table 4.1: Summary of mathematical notation. Lower case letters are used for points and upper case letters are used for buckets and regions. Additional conventions are introduced in Table 4.2.

operate at the level of buckets. The space of buckets mined by the data mining algorithm is then visualized using color maps. The color of each bucket is the confidence that the points (configurations) that map to this bucket exhibit acceptable average performance.

4.2.1 The First Level of Aggregation: Points

Table 4.1 summarizes some of the syntactic conventions used in this chapter. Mathematically, we can think of the WCDMA simulation as estimating the mean $E[x_k]$ of a random variable x_k with some (unknown) distribution [37] (x_k is one when the information bit is decoded in error or zero when it is decoded correctly). The BER \hat{x}_{kj} , $1 \leq j \leq n_k$, output by the simulation is an unbiased estimate of the BEP $E[x_k]$ of the simulated configuration c_k . The distribution of x_k is analytically inconvenient. Instead of building a detailed stochastic model of the simulation, we choose to work with the simpler distribution of the BEP $b_k = E[x_k] \approx \hat{x}_{kj}$. The distribution of b_k is approximately Gaussian due to the Central Limit Theorem. We assume that the number of frames per estimate \hat{x}_{kj} is ‘large enough’ so that the Lindeberg condition is satisfied, that the variance of \hat{x}_{kj} is finite, and that $\{\hat{x}_{kj}\}_{j=1}^{n_k}$ are i.i.d. We say that $E[b_k] = E[E[x_k]]$ is the *expected BEP* of configuration c_k under Rayleigh fading.

4.2.2 The Second Level of Aggregation: Buckets

Let us now aggregate several points (i.e., random variables) into one bucket. The purpose of this aggregation is to reduce data dimensionality to a size that is easy to visualize, usually one or two dimensions. The basic idea is to linearly average all points that map to the same bucket but we must do so carefully, in order to preserve a meaningful statistical interpretation. Let $\{b_k\}_{k=1}^n$ be Gaussian random variables with means $\{E[b_k]\}_{k=1}^n$ and variances $\{\sigma_k^2\}_{k=1}^n$. As in the previous paragraph, let each such variable b_k be the BEP of some configuration c_k , $1 \leq k \leq n$. For bucket C , define a *bucket (mixture) random variable* B as

$$B = \sum_{k=1}^n p_k b_k,$$

where $\{p_k\}_{k=1}^n$ are (positive) constant weights of $\{b_k\}_{k=1}^n$. It is convenient to make $\{p_k\}_{k=1}^n$ the probabilities of occurrence of the configurations $\{c_k\}_{k=1}^n$ in the dataset being analyzed. This setup underlines the dependence of the outputs on the distribution of the inputs and frees the user from having to provide

values for the constants $\{p_k\}_{k=1}^n$. It is well known that, as long as $\{b_k\}_{k=1}^n$ are *mutually independent* and Gaussian with means $\{E[b_k]\}_{k=1}^n$ and variances $\{\sigma_k^2\}_{k=1}^n$, B is Gaussian with mean $E[B] = \sum_{k=1}^n p_k E[b_k]$ and variance $\Sigma^2 = \sum_{k=1}^n p_k^2 \sigma_k^2$ [16]. The expected value $E[B]$ of the random variable B can be viewed as the expected BEP of bucket $C = \{c_1, c_2, \dots, c_n\}$ in a Rayleigh fading environment, conditional on the (discrete) distribution of the configurations in C .

The values $\{p_k\}_{k=1}^n$ are what the statisticians call *prior probabilities*. For most purposes of this chapter, we simply estimate $\{p_k\}_{k=1}^n$ from available data. These values are explicitly or implicitly constructed during experiment design and we assume that they remain constant during experiment analysis. However, one can collect additional data as long as doing so does not change $\{p_k\}_{k=1}^n$. Prior probabilities can come from a number of sources: channel sounding measurements, propagation simulations, hardware and budget constraints, or even educated guesses by wireless system designers. The rest of the chapter silently assumes that the values $\{p_k\}_{k=1}^n$ have been established beforehand. It is important to remember that even though the prior probabilities are for the most part transparent to the analysis presented here, they nonetheless always exist and all conclusions of data analysis are made conditional on the prior probabilities.

This discussion of $\{p_k\}_{k=1}^n$ can be interpreted as a deferral of the exact definition of B until experiment setup, or as parameterization of the analysis procedure. A natural question is whether or not this level of parameterization is sufficient. It is sufficient for the purposes of this chapter but, strictly speaking, the interrelations between $\{b_k\}_{k=1}^n$ should also be defined during experiment setup. Mutual independence of $\{b_k\}_{k=1}^n$ is a simplifying assumption and it might be desirable to model interactions between $\{b_k\}_{k=1}^n$ in practice. This implies adding covariance terms to Σ^2 and re-thinking the distribution of B . Such analysis is necessarily specific to a particular experiment. For the sake of simplicity, the rest of this chapter assumes mutual independence of variables in a given bucket.

4.2.3 Confidence Estimation

Point and bucket estimates of the expected BEP are meaningful performance metrics for wireless systems. Let us also estimate our confidence in these estimates. Confidence analysis enables wireless system designers to make more practical claims than point estimates alone. A statement of the form ‘this configuration will exhibit acceptable performance in 95% of the cases’ is often preferable to a statement of the form ‘the expected BEP of this configuration is approximately 5×10^{-4} ’. More precisely, we say that configuration c_k *exhibits acceptable performance* when the expected BEP $E[b_k]$ of configuration c_k is below some fixed threshold T . This statement is conditional on the temporal simulation assumptions, i.e., Rayleigh fading. Standard values for T are 10^{-3} for voice quality systems and 10^{-6} for data quality systems. Likewise, we say that bucket C (a subspace of configurations) *exhibits acceptable average performance* when the expected BEP $E[B]$ of bucket C is below some fixed threshold T . This statement is conditional on both the temporal simulation assumptions and the distribution of configurations $\{c_k\}_{k=1}^n$ in the bucket (the prior probabilities).

The confidence that configuration c_k (resp. bucket C) exhibits acceptable (average) performance is $P(E[b_k] < T)$ (resp. $P(E[B] < T)$). Since b_k and B are Gaussian, these probabilities can be estimated as

$$P(E[b_k] < T) \approx F_{n_k-1} \left(\frac{T - \hat{b}_k}{\hat{\sigma}_k / \sqrt{n_k}} \right), \quad P(E[B] < T) \approx F_{N-1} \left(\frac{T - \hat{B}}{\hat{\Sigma} / \sqrt{N}} \right),$$

where $F_{N-1}(\cdot)$ is the CDF of the Student t distribution with $N - 1$ degrees of freedom and n_k and N are the sample sizes for configuration c_k and bucket C , respectively. For configuration c_k ,

$$\hat{b}_k = \frac{1}{n_k} \sum_{j=1}^{n_k} \hat{x}_{kj}, \quad \hat{\sigma}_k^2 = \frac{1}{(n_k - 1)} \sum_{j=1}^{n_k} (\hat{x}_{kj} - \hat{b}_k)^2,$$

where \hat{b}_k and $\hat{\sigma}_k^2$ are the estimates of the expected BEP and the BEP variance at point c_k , $n_k \geq 2$ is sample size, and $\{\hat{x}_{kj}\}_{j=1}^{n_k}$ are sample values. For bucket C , we substitute point estimates into $E[B] = \sum_{k=1}^n p_k E[b_k]$ and $\Sigma^2 = \sum_{k=1}^n p_k^2 \sigma_k^2$ to obtain

$$\hat{B} = \sum_{k=1}^n \hat{p}_k \hat{b}_k, \quad \hat{\Sigma}^2 = \sum_{k=1}^n \hat{p}_k^2 \hat{\sigma}_k^2, \quad N = \min_{1 \leq k \leq n} n_k,$$

where \hat{B} and $\hat{\Sigma}^2$ are the estimates of the expected BEP and the BEP variance at bucket C , N serves the role of ‘bucket sample size’, and $\{\hat{p}_k\}_{k=1}^n$ are the prior probabilities estimated from the dataset as $\hat{p}_k = n_k / \sum_{i=1}^n n_i$. Observe that

$$\hat{B} = \sum_{k=1}^n \hat{p}_k \hat{b}_k = \frac{1}{\sum_{k=1}^n n_k} \sum_{k=1}^n \sum_{j=1}^{n_k} \hat{x}_{kj}$$

is exactly the sample mean of all observations in the bucket, but $\hat{\Sigma}^2$ is *not* the variance and N is *not* the size of this sample. This is the case because $\{\{\hat{x}_{kj}\}_{j=1}^{n_k}\}_{k=1}^n$ are not i.i.d. samples from the mixture distribution of B —they are samples from the constituent distributions of $\{b_k\}_{k=1}^n$.

4.3 Extended Example

Let us now apply the techniques developed so far to analyze the performance of a space of configurations. The wireless systems under consideration employ WCDMA technology with two-branch STTD and 1/2 rate convolutional coding. We require that the transmitter has two antennas (branches) separated by a distance large enough for their signals to be uncorrelated, but small enough for the mean path losses and impulse responses of their channels to be approximately equal at receiver locations of interest. We assume Rayleigh flat fading channels, which is reasonable for indoor applications in the ISM and UNII carrier frequency bands (2.4 and 5.2 GHz, respectively). The goal is to study the effect of power imbalance between the branches on the BEP of the configurations across a wide range of average SNRs.

This section presents a number of plots that summarize simulated BERs. We also outline the process of statistically significant sampling of the configuration space. The next section develops a data mining methodology that solves a practically important problem: given a dataset similar to the one presented next, find a region of the configuration space where we can confidently claim that configurations will exhibit acceptable (average) performance.

Let us begin with an initial sample of the configuration space, as shown in Figure 4.2 (top). This figure shows the simulated BER as a 2D function $\hat{f}(S_1, S_2)$ of the average branch bit energy-to-noise ratios (SNRs) S_1 and S_2 , in dB. The parallel simulation ran for three days on 120 machines (AMD Athlon 1.0 GHz) at a speed of approximately 2.5 points per machine per day. 10000 frames, or 800000 information bits, were simulated for each of the 820 points $S_2 = 3, 4, \dots, 42$; $S_1 = 3, 4, \dots, S_2$. Since $\hat{f}(S_1, S_2)$ is symmetric [21], we show $M = 1600$ points $\{c_k\}_{k=1}^M$ for a full cross-product of S_1 and S_2 .

Wireless system designers are more accustomed to 1D slices of the configuration space, e.g., the ones shown in Figure 4.3. Define the *branch power imbalance factor*

$$\alpha = 10^{-0.1|S_1 - S_2|},$$

where S_1 and S_2 are the average SNRs of the branches, in dB. (This definition applies as long as the mean path losses of the branches are equal.) By definition, $0 \leq \alpha \leq 1$, where zero corresponds to a total

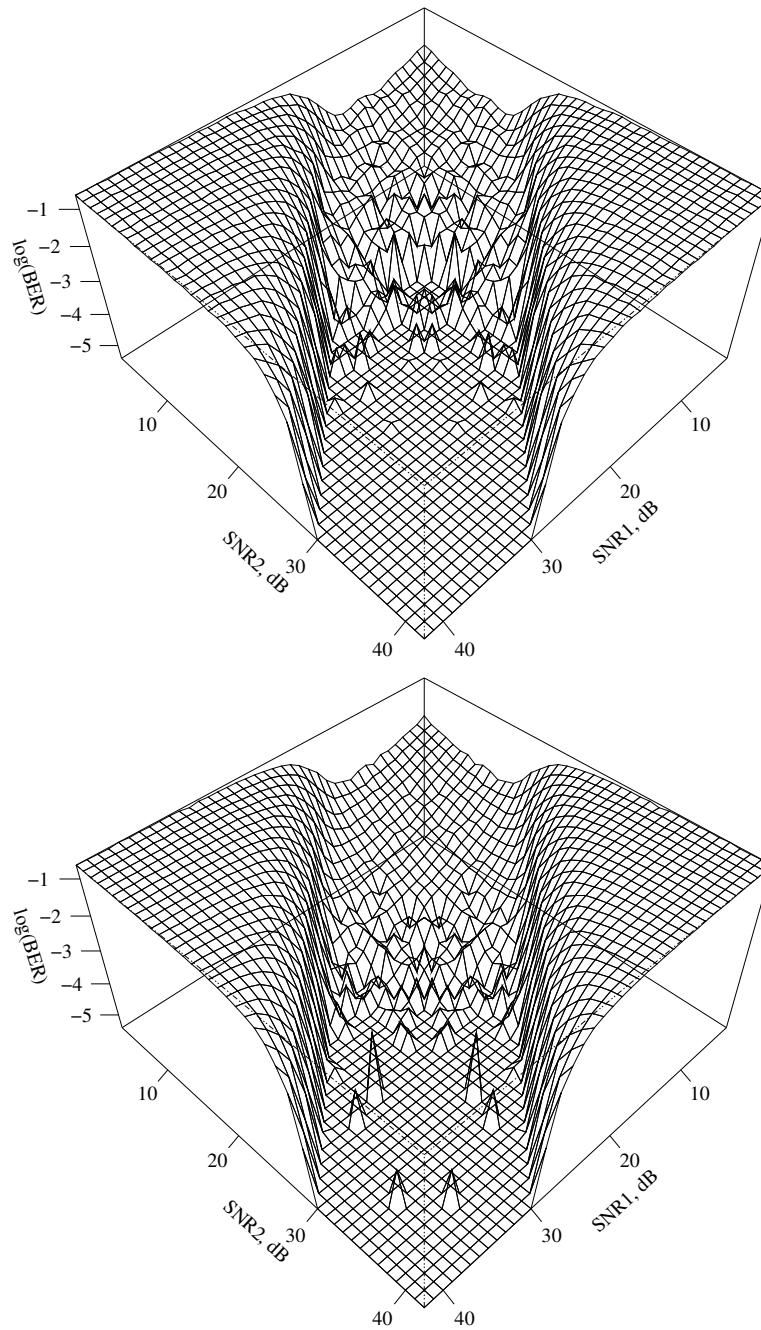


Figure 4.2: (top) Estimates of the BEPs for a space of configurations $\{c_k\}_{k=1}^M$ ($M = 1600$ points at 10000 frames per point). The X and Y axes are the average SNRs of the branches (in dB). The Z axis is the (base ten) logarithm of the simulated BER. These estimates are not statistically significant. (bottom) Statistically significant estimates $\{\hat{b}_k\}_{k=1}^M$ of the expected BEPs $\{E[b_k]\}_{k=1}^M$ for the same space of configurations $\{c_k\}_{k=1}^M$. For the most part, we are 90% confident that the estimated expected BEP lies within 10% of its true value. See text for exceptions.

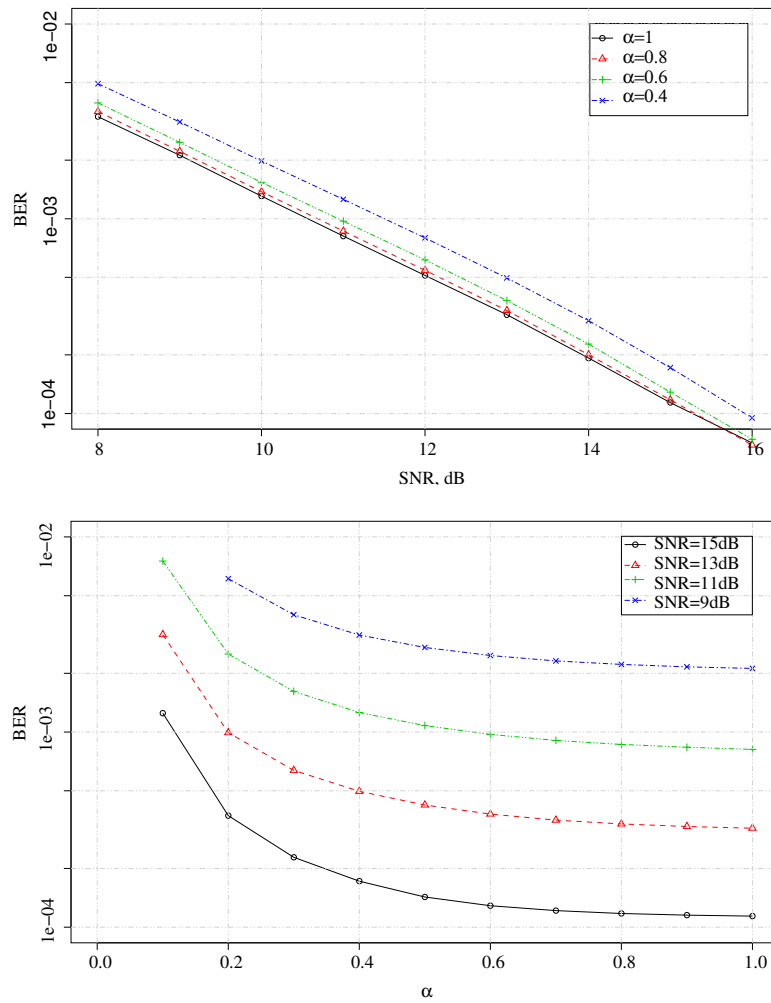


Figure 4.3: 1D slices of the configuration space $\{c_k\}_{k=1}^M$ with fixed branch power imbalance factor $\alpha = 10^{-0.1|S_1-S_2|}$ and varying effective SNR $S = 10 \log_{10} \left(\frac{10^{0.1S_1} + 10^{0.1S_2}}{2} \right)$ (top), and fixed effective SNR S and varying branch power imbalance factor α (bottom). These slices were computed from the surface fit onto the data in Figure 4.2 (bottom). The entire fitted surface is shown in Figure 4.4.

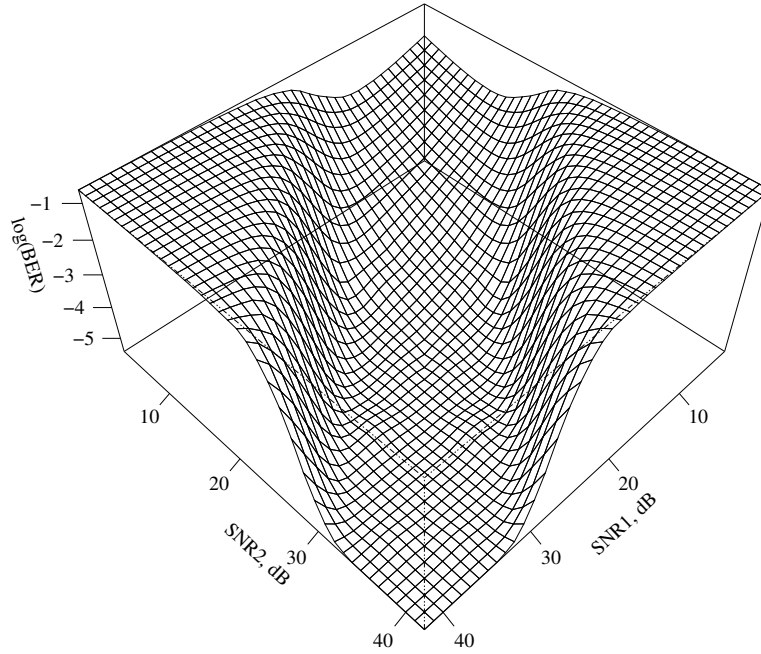


Figure 4.4: A surface fitted onto the statistically significant results in Figure 4.2 (bottom). We used a local linear least squares regression with a 5% neighborhood and tricubic weighting. This procedure was chosen because it can approximate the relatively steep edge of the tolerance region. See [33] for details.

malfunction of one of the branches and one corresponds to a perfect balance of branch powers. The graphs in Figure 4.3 were obtained by fixing α and varying the *effective SNR*

$$S = 10 \log_{10} \left((10^{0.1S_1} + 10^{0.1S_2})/2 \right),$$

in dB (top), and fixing the effective SNR and varying α (bottom). (Note that fixing the effective SNR is equivalent to fixing total transmitter power.) The sample of configurations in Figure 4.2 (bottom) does not contain the exact points for typical slices, so we used a fitted surface in Figure 4.4 to approximate the BERs for the slices in Figure 4.3. We choose to work with the axes S_1, S_2 in Figure 4.2 because it simplifies the discussion later.

What can be gathered from Figure 4.2 (top)? The deep valley along the diagonal is due to the fact that, provided that the effective SNR is fixed, we expect the BEP to be smallest when the branch power is balanced ($S_1 = S_2, \alpha = 1$) [21]. Somewhat less expected were (a) the wide *tolerance region* where $|S_1 - S_2|$ is large (up to 12 dB) but the BER is still small, (b) a very sharp decline in performance at the edge of the tolerance region, and (c) a region of high local variability in the upper part of the diagonal. The surface is truncated at

$$\min_{1 \leq k \leq M} \{\hat{b}_k\} = 3.75 \times 10^{-6}$$

because smaller estimates of the (expected) BEP require an enormous computation time due to the convergence properties of Monte Carlo Estimation (more on this below).

4.3.1 Statistically Significant Sampling Methodology

The initial sample looks reasonable and uncovers interesting trends in system performance, but it does not contain enough information to make statistically significant claims. Estimating the probability that a configuration exhibits acceptable average performance requires several samples per point c_k . The simulation is computationally expensive and different regions of the configuration space exhibit different variability. Therefore, we must define tight stopping criteria for sampling. Figure 4.2 (bottom) shows the output obtained with the following (per point c_k) stopping criteria. The criteria are designed to achieve high estimation accuracy.

1. Sampling $\{\hat{x}_{kj}\}$ stops when the relative error in the estimate \hat{b}_k of the expected BEP $E[b_k]$ is smaller than the *relative accuracy threshold* $\beta = 0.1$ times the current estimate \hat{b}_k , at a $\gamma = 0.9$ confidence level, i.e., when

$$P(|E[b_k] - \hat{b}_k| < \beta \hat{b}_k) \geq \gamma.$$

We required $n_k \geq 2$ samples to obtain an estimate $\hat{\sigma}_k^2$ of the BEP variance σ_k^2 . Notice that the target is the relative error, not the absolute error, because the range of $\{\hat{b}_k\}_{k=1}^M$ in the configuration space spans four orders of magnitude. Therefore, absolute error measures are misleading.

2. Sampling $\{\hat{x}_{kj}\}$ also stops when we can say, with confidence $\gamma = 0.9$, that the expected BEP $E[b_k]$ is below the *sampling threshold* $t = 10^{-4}$, i.e., when

$$P(E[b_k] < t) \geq \gamma.$$

This work considers voice quality applications, so the exact value of the expected BEP is irrelevant as long as it is smaller than the performance threshold $T = 10^{-3}$. The sampling threshold t was set to an order of magnitude below the performance threshold T to avoid large approximation error of a fitted surface near T .

3. Finally, sampling $\{\hat{x}_{kj}\}$ stops when more than 50 samples of 10000 frames each are required to satisfy either of the previous rules. This rule fired in 5% of the cases, all at the boundary of the tolerance region and most in mid diagonal.

Altogether, 5154 samples were collected for an average of 6.3 samples per point. Needless to say, the computational expense of such sampling remains too high for practical applications. However, a large number of samples is desirable to validate our framework. As we shall see later, our framework requires fewer samples. Let us now look at the data in more detail.

4.3.2 Results of Statistically Significant Sampling

We assumed that the BEPs $\{b_k\}_{k=1}^M$ are Gaussian. Intuitively, we are simulating a large number of information bits (800000) per BEP estimate \hat{x}_{kj} , so the Lindeberg condition for the Central Limit Theorem should hold. Figure 4.5 shows empirical evidence that this is the case. We have arbitrarily chosen one point among those with 20–30 sample values $\{\hat{x}_{kj}\}_{j=1}^{n_k}$ and plotted the empirical CDF of this sample against that of the Gaussian distribution with the mean equal to sample mean \hat{b}_k and the variance equal to sample variance $\hat{\sigma}_k^2$. The curves are close to each other and the Shapiro-Wilk test yields $W = 0.98$ ($0 \leq W \leq 1$) and p -value of 0.88. Other points also demonstrate similar curves and high values of W , but p -values vary significantly. This dataset contains sufficient samples to estimate $\{E[b_k]\}_{k=1}^M$ with high relative accuracy, but 6.3 samples per point are insufficient to formally justify a Gaussian assumption.

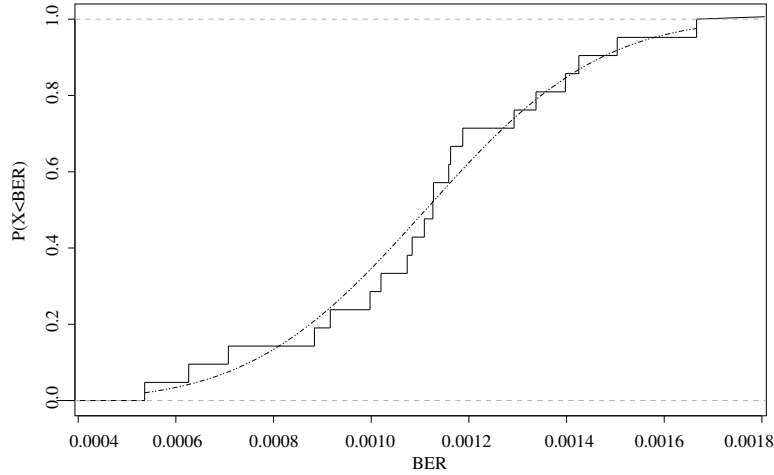


Figure 4.5: Empirical CDF of 21 samples for a randomly chosen point vs. that of the Gaussian distribution with appropriate mean and variance.

It is also instructive to see some measure of how the sample variance is distributed across the configuration space. Figures 4.6 and 4.7 show sample sizes and sample standard deviation-to-mean ratios for the samples in Figure 4.2 (recall that we prefer relative measures because the range of $\{\hat{b}_k\}_{k=1}^M$ is large). Both figures indicate high variance around the boundary of the tolerance region. This is not surprising because the edges of the tolerance region are relatively steep. Figure 4.7 also shows relatively high variance at some points inside the tolerance region. This is because the simulation achieved the sampling threshold $t = 10^{-4}$ and stopped before it achieved the relative accuracy threshold $\beta = 0.1$. Knowing this, one would expect a larger relative variance in the tolerance region. Let us examine why this is not the case.

We treat the BEP as a continuous Gaussian random variable b_k , but all sample values $\{\hat{x}_{kj}\}_{j=1}^{n_k}$ are discrete—they are ratios of two integers, the number of errors and the number of bits simulated. The simulation may not detect any bit errors when the expected BEP $E[b_k]$ is relatively small (e.g., one error in the number of bits simulated). Since no channel is perfect, zero is too optimistic an estimate for the expected BEP. Instead, we conservatively assume that at least three bit errors have been detected. This is why the smallest estimate \hat{b}_k of $E[b_k]$ is $3/800000 = 3.75 \times 10^{-6}$. However, using any constant cutoff prevents us from estimating the variance σ_k^2 . We would need to simulate a large number of frames to estimate σ_k^2 when the expected BEP is small. Instead, we can empirically show that the probability that the expected BEP is smaller than the performance threshold $T = 10^{-3}$ is close to one. Let $\hat{b}_k = 3.75 \times 10^{-6}$ be the sample mean, $n_k = 2$ be the sample size, and σ_k^2 be the BEP variance at point c_k where two independent simulations detected three or fewer bit errors each. Sampling $\{\hat{x}_{kj}\}$ will stop because sample variance is zero, so the first stopping rule applies.

We need to show that sampling can indeed stop, i.e., that the probability that the expected BEP is below the performance threshold T is

$$P(E[b_k] < T) \approx F_{n_k-1} \left(\frac{T - \hat{b}_k}{\sigma_k / \sqrt{n_k}} \right) \geq 0.995.$$

This statement can only be false when $(T - \hat{b}_k)\sqrt{n_k}/\sigma_k \leq 64$, or $\sigma_k \geq 2.2 \times 10^{-5}$, almost an order of magnitude bigger than the conservative estimate \hat{b}_k of the expected BEP $E[b_k]$. This is unlikely because

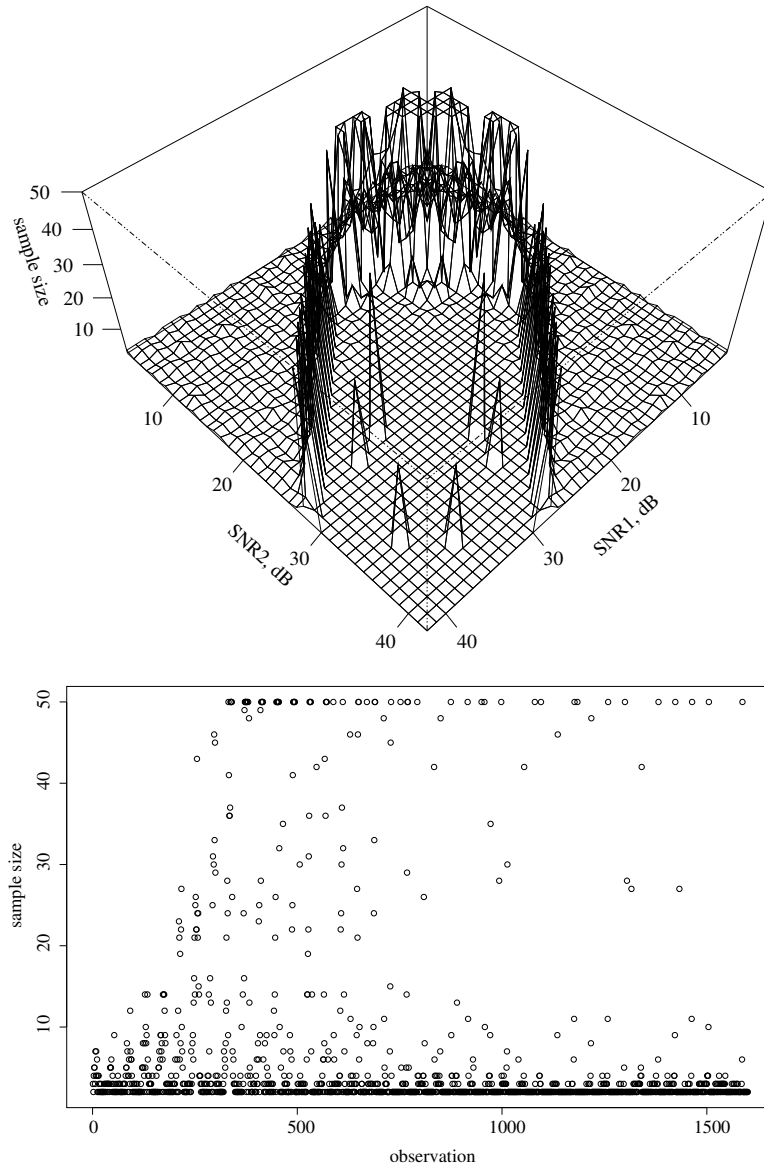


Figure 4.6: Sample sizes for Figure 4.2 (bottom). The top part shows the perspective plot and the bottom part shows the scatter plot.

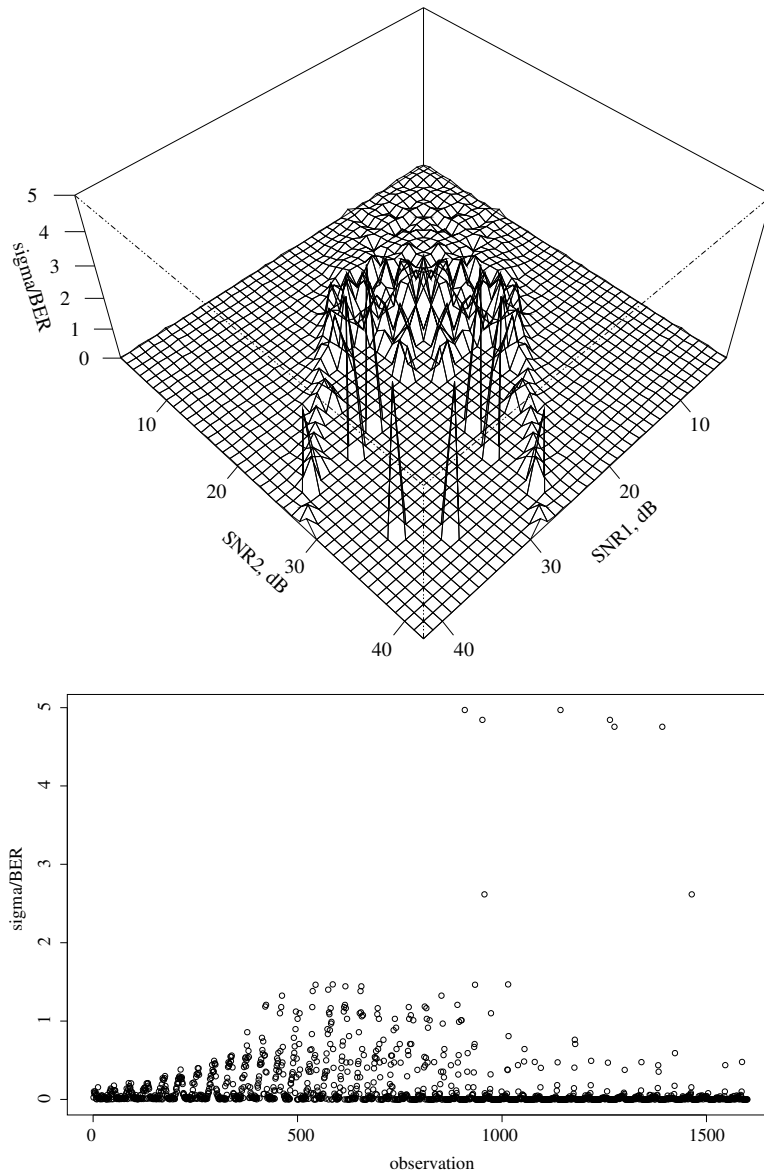


Figure 4.7: Sample standard deviation-to-mean ratios for Figure 4.2 (bottom). The top part shows the perspective plot and the bottom part shows the scatter plot.

X, Y	parameters that partition the point space into buckets
M_X, M_Y	X and Y dimensions of the bucket space
$M = M_X \times M_Y$	number of buckets in the bucket space
D_X, D_Y	domains of X and Y
$\eta(m)$	number of buckets in region R_m
$C_{\kappa(m,i)}$	i -th bucket in region R_m , $1 \leq i \leq \eta(m)$
$x_{\kappa(m,i)}, y_{\kappa(m,i)}$	X and Y values for bucket $C_{\kappa(m,i)}$

Table 4.2: Summary of region notation. Also see Table 4.1.

Figure 4.7 (bottom) shows that the sample standard deviation rarely exceeds the sample mean even by half an order of magnitude. In other words, we do not have accurate estimates for variance σ_k^2 in the tolerance region. However, we can still reasonably conclude that configurations exhibit acceptable performance in this region.

4.4 The Third Level of Aggregation: Regions

Consider a set of buckets $\{C_k\}_{k=1}^M$ with corresponding random variables $\{B_k\}_{k=1}^M$. Given a number of sample values, the framework developed in Section 4.2 allows us to estimate the probabilities $\{P(E[B_k] < T)\}_{k=1}^M$ that buckets $\{C_k\}_{k=1}^M$ exhibit acceptable average performance. (All arguments about buckets equally apply to points because a point is a special case of a bucket.) This section is concerned with finding an optimal subset of random variables from among $\{B_k\}_{k=1}^M$. This optimal subset corresponds to an optimal region of a 2D bucket space. We would like to find a sufficiently large admissible region R_m such that we are sufficiently confident that buckets in R_m exhibit acceptable average performance.

There are many ways to define admissibility and we are interested in adopting a definition that is both meaningful in the wireless domain and permits effective data mining algorithms. Among a space of such admissible regions, we can define different optimality criteria and data mining then reduces to searching within this space. In this chapter, a region R_m is admissible when it has a particular type of shape. We will explore three different criteria for the mining of optimal regions; the algorithms and these criteria are based on the work of Fukuda et al. [52] and have been adapted to the problem of mining simulation data in this chapter.

Additional notation relating buckets to regions is introduced in Table 4.2. Let X and Y be two discrete parameters to the temporal (e.g., WCDMA) simulations such that X and Y partition the point space into disjoint buckets $\{C_k\}_{k=1}^M$. More precisely, let X, Y have ordinal domains D_X, D_Y , let $|D_X| = M_X, |D_Y| = M_Y, |D_X||D_Y| = M$, and assume that the map $\rho : D_X \times D_Y \rightarrow \{C_k\}_{k=1}^M$ is bijective. In other words, X and Y define a discrete 2D space of buckets. Since the domains of X and Y are ordinal, this space is easily visualized as a 2D color map or a 3D perspective plot.

For example, the average SNRs S_1 and S_2 in the previous section partition the space of configurations into buckets. Both S_1 and S_2 vary from 3 to 42 in steps of 1 (in dB), so $M_X = M_Y = 40$ and $M = 40 \times 40 = 1600$ (recall, from Section 4.3, that only 820 of these points were simulated and the remaining ones were symmetrically reflected). Furthermore, the domains of S_1 and S_2 are ordinal because the values of S_1 and S_2 are directly related to the powers of the transmitter antennas. In this case, the buckets are simply the points in the space of configurations. In general, buckets can be convex combinations of points, as detailed in Section 4.2. Recall that we defined the color of a bucket as the probability that the bucket exhibits acceptable average performance. Figure 4.8 shows these ‘colors’ as a perspective plot for the STTD

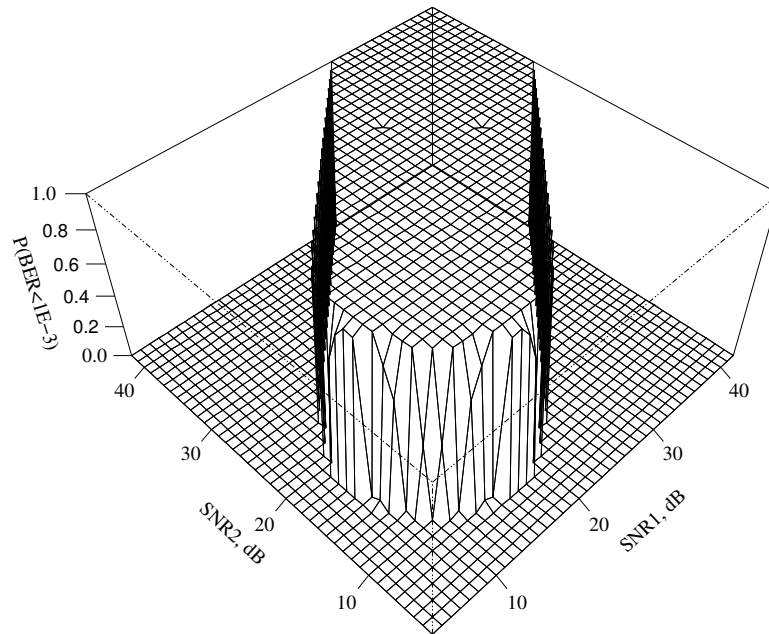


Figure 4.8: Probabilities $\{P(E[b_k] < T)\}_{k=1}^M$ that configurations $\{c_k\}_{k=1}^M$ exhibit acceptable performance with respect to the performance threshold $T = 10^{-3}$ (voice quality system). This perspective plot corresponds to the STTD dataset in Figure 4.2 (bottom). The axes S_1 and S_2 are rotated 180 degrees counter-clockwise to provide a better view of the surface.

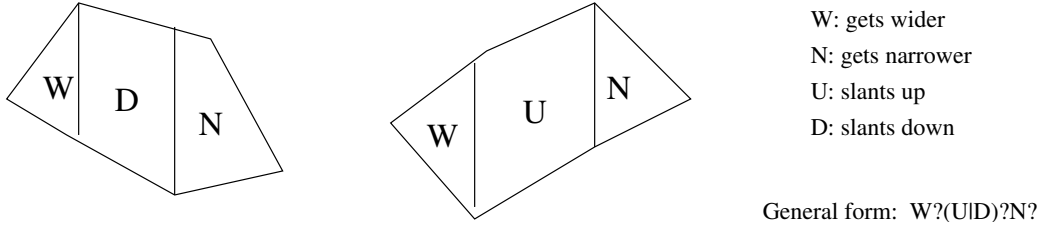


Figure 4.9: Some types of admissible (connected rectilinear) regions. When we look at an admissible region from left to right, its upper boundary must first increase and then decrease monotonically, and its lower boundary must first decrease and then increase monotonically.

example.

4.4.1 Region Shape

Consider regions (subsets) of buckets in the bucket space. If the shape of these regions is unconstrained, there are 2^M possible regions $\{R_m\}_{m=1}^{2^M}$. Let region R_m , $1 \leq m \leq 2^M$, consist of buckets $\{C_{\kappa(m,i)}\}_{i=1}^{\eta(m)}$, where $\eta(m)$, $1 \leq m \leq 2^M$, is a mapping from region number m to the number of buckets in this region, and $\kappa(m, i)$, $1 \leq m \leq 2^M$, $1 \leq i \leq \eta(m)$, is a mapping from region number m and bucket number i within region R_m to bucket number k , $1 \leq k \leq M$, that we use to subscript buckets $\{C_k\}_{k=1}^M$. The exact definitions of $\eta(m)$ and $\kappa(m, i)$ are not important as long as they generate all possible regions (subsets) $\{R_m\}_{m=1}^{2^M}$.

The shape of admissible regions should be constrained because unconstrained regions are hard to interpret and tend to overfit the training data. Besides, the problem of selecting an optimal unconstrained region is computationally intractable—all 2^M possible regions must be considered, where $M = 1600$ in the STTD example. The region shape can be constrained in a number of different ways (rectangular, x-monotone, etc.). Our restrictions on region shape are discussed next.

Without loss of generality, assume that $D_X = \{1, 2, \dots, M_X\}$ and $D_Y = \{1, 2, \dots, M_Y\}$. Intuitively, region R_m is rectilinear when its intersection with any horizontal or vertical line is connected. More formally, region R_m is *rectilinear* if and only if whenever buckets $C_{\kappa(m,i)}$ at $(x_{\kappa(m,i)}, y_{\kappa(m,i)})$ and $C_{\kappa(m,j)}$ at $(x_{\kappa(m,j)}, y_{\kappa(m,j)})$ are both in R_m , then (a) $x_{\kappa(m,i)} = x_{\kappa(m,j)}$ implies that each bucket $C_{\kappa(m,l)}$ where

$$(y_{\kappa(m,l)} - y_{\kappa(m,i)})(y_{\kappa(m,l)} - y_{\kappa(m,j)}) < 0$$

is also in R_m , and (b) likewise for $y_{\kappa(m,i)} = y_{\kappa(m,j)}$. We use Manhattan geometry to define connectedness. Region R_m is *connected* if and only if for every pair of buckets $C_{\kappa(m,i)}$ and $C_{\kappa(m,j)}$ in R_m there exists a sequence of buckets

$$C_{\kappa(m,i)} = C_{\kappa(m,l_1)}, C_{\kappa(m,l_2)}, \dots, C_{\kappa(m,l_n)} = C_{\kappa(m,j)}$$

in R_m such that for every $1 \leq k < n$

$$|x_{\kappa(m,l_k)} - x_{\kappa(m,l_{k+1})}| + |y_{\kappa(m,l_k)} - y_{\kappa(m,l_{k+1})}| = 1.$$

Furthermore, we say that region R_m is *admissible* if it is both rectilinear and connected.

This definition of admissibility can be viewed as a relaxed definition of convexity. Geometrically, it is easy to see that region R_m is admissible if and only if, when we look at R_m from left to right, its upper

boundary first increases and then decreases monotonically (a pseudoconcave function), and its lower boundary first decreases and then increases monotonically (a pseudoconvex function). In other words, the region boundary need not be strictly convex or strictly concave, but it must be pseudoconvex or pseudoconcave. Admissible regions are informally summarized in Figure 4.9. All admissible regions are composed of regions of four primitive types: W (region gets wider from left to right), N (region gets narrower), U (region slants up), and D (region slants down). Twelve combinations of these types yield all types of admissible regions: W, WU, WUN, WD, WDN, WN, UN, DN, U, D, N, and the empty region.

Our choice of connected rectilinear regions is due to primarily heuristic considerations. These considerations are commonly applicable, but must be re-evaluated for each study. Both the connectedness and the rectilinearity restrictions can be justified for the STTD example (see next section). In general, it is easy to justify connectedness, but hard to justify rectilinearity. We advocate the use of connected rectilinear regions primarily because this shape is resistant to noise in the sample, not because we can analytically show that the region boundary is rectilinear. In data mining, the choice of region shape is most commonly dictated by the desired tradeoff between bias and variance [33]. Regions with flexible shape exhibit small bias (they can fit any data) but high variance (they can be overly sensitive to a particular dataset). Regions with rigid shape exhibit high bias but small variance. Connected rectilinear regions provide a reasonable tradeoff between bias and variance for many applications.

4.4.2 Region ‘Goodness’

Another prerequisite to finding regions with the desired properties is a definition of region ‘goodness’. Let us map bucket confidence $P(E[B_{\kappa(m,i)}] < T)$ to a discrete range $[0 \dots 1000]$ and define the *hit of bucket* $C_{\kappa(m,i)}$ as

$$h_{\kappa(m,i)} = \lfloor 1000P(E[B_{\kappa(m,i)}] < T) + 0.5 \rfloor$$

($\lfloor X \rfloor$ denotes the largest integer that does not exceed X), the *support of bucket* $C_{\kappa(m,i)}$ as

$$s_{\kappa(m,i)} = 1000$$

(this constant was chosen to make the discretization error reasonably small), the *hit of region* R_m as

$$H_m = \sum_{i=1}^{\eta(m)} h_{\kappa(m,i)},$$

and the *support of region* R_m as

$$S_m = \sum_{i=1}^{\eta(m)} s_{\kappa(m,i)} = 1000\eta(m).$$

The key to efficient computation of optimized-confidence and optimized-support admissible regions is the definition of region confidence as

$$\Theta_m = H_m/S_m,$$

where H_m is the hit and S_m is the support of region R_m . Let us explore the implications of these definitions in more detail.

Model-Based and Model-Free Analyses

Suppose, $n_{\kappa(m,i)} = 6$ samples have been collected for bucket $C_{\kappa(m,i)}$ that consists of a single point. Let the sample mean be $\hat{B}_{\kappa(m,i)} = 5 \times 10^{-4}$ and the sample standard deviation be $\hat{\Sigma}_{\kappa(m,i)} = 8.87 \times 10^{-4}$. Furthermore, suppose that five of these samples have the BER below 10^{-3} and one has the BER above 10^{-3} . Then,

$$P(E[B_{\kappa(m,i)}] < T) \approx F_5 \left(\frac{10^{-3} - 5 \times 10^{-4}}{8.87 \times 10^{-4} / \sqrt{6}} \right) \approx 0.887.$$

A purely model-free approach would interpret the above simulation results as ‘bucket $C_{\kappa(m,i)}$ will exhibit acceptable average performance in 5 out of 6 cases.’ A strongly model-based approach would interpret the simulation results as ‘we are 88.7% confident that bucket $C_{\kappa(m,i)}$ exhibits acceptable average performance.’ Our interpretation lies between the model-based approach and a model-free approach and posits that ‘bucket $C_{\kappa(m,i)}$ will exhibit acceptable average performance in 887 out of 1000 cases.’ These interpretations provide confidence estimates under different simplifying assumptions.

The model-free interpretation does not take either sample variance or sample distribution into account. This interpretation is only reliable for a sufficiently large number of samples, which is a luxury in our application. Our middle-ground interpretation explicitly accounts for sample variance and sample distribution. When sample size is small, our interpretation provides a statistically valid estimate of confidence that the bucket exhibits acceptable average performance. For a single bucket, this interpretation is as good as a strongly model-based interpretation, modulo a reasonably small discretization error. However, our interpretation diverges from the model-based interpretation at the region level.

A strongly model-based analysis procedure would define a region random variable

$$Q_m = \frac{1}{W_m} \sum_{i=1}^{\eta(m)} w_{\kappa(m,i)} B_{\kappa(m,i)},$$

where $\{B_{\kappa(m,i)}\}_{i=1}^{\eta(m)}$ are bucket random variables, $\{w_{\kappa(m,i)}\}_{i=1}^{\eta(m)}$ are *a priori* (positive) constant weights, and

$$W_m = \sum_{i=1}^{\eta(m)} w_{\kappa(m,i)}$$

is a normalization factor that maps these weights to probabilities of bucket occurrence in the region. A procedure similar to that in Section 4.2 would then be used to estimate $P(E[Q_m] < T)$ for a threshold T . The result of this calculation can be interpreted as the probability that region R_m exhibits acceptable average performance, conditional on the temporal simulation assumptions, the bucketing prior probabilities, and the region prior probabilities. However, as we shall see later, this definition of region confidence violates a property that permits an efficient data mining algorithm.

We think of region confidence in terms of average bucket confidence over the whole region, namely,

$$\Theta_m \approx \frac{1}{\eta(m)} \sum_{i=1}^{\eta(m)} P(E[B_{\kappa(m,i)}] < T).$$

(If region size $\eta(m)$ is large enough, we can reasonably expect the discretization errors to cancel each other.) This interpretation of Θ_m does not correspond to the strongly model-based probability that region R_m exhibits acceptable average performance. Instead, we define a region random variable P_m as the probability

that any bucket $C_{\kappa(m,i)}$ in region R_m exhibits acceptable average performance. Then, we estimate the expected value $E[P_m]$ across the region R_m by the sample mean $\hat{P}_m \approx \Theta_m$ of estimates of bucket confidences $\{P(E[B_{\kappa(m,i)}] < T)\}_{i=1}^{\eta(m)}$.

How do these two definitions relate to each other? It is easy to show that they are equivalent under very restrictive assumptions. Basically, we are assuming that the buckets are mutually independent, that population variance is small, and that the region is consistent, i.e., ‘good’ and ‘bad’ buckets are never mixed in the same region. Let bucket random variables $\{B_{\kappa(m,i)}\}_{i=1}^{\eta(m)}$ be mutually independent, let the estimates $\{\hat{\Sigma}_{\kappa(m,i)}^2\}_{i=1}^{\eta(m)}$ of bucket variances be approximately equal to zero, and let the estimates $\{\hat{B}_{\kappa(m,i)}\}_{i=1}^{\eta(m)}$ of bucket expected BEPs be either all greater than the performance threshold T or all smaller than the performance threshold T (i.e., all $\{T - \hat{B}_{\kappa(m,i)}\}_{i=1}^{\eta(m)}$ have the same sign). Then, bucket confidences

$$P(E[B_{\kappa(m,i)}] < T) \approx F_{n_{\kappa(m,i)}-1} \left(\frac{T - \hat{B}_{\kappa(m,i)}}{\hat{\Sigma}_{\kappa(m,i)} / \sqrt{n_{\kappa(m,i)}}} \right),$$

$1 \leq i \leq \eta(m)$, will be either all approximately equal to zero ($\hat{B}_{\kappa(m,i)} > T$), or all approximately equal to one ($\hat{B}_{\kappa(m,i)} < T$). Therefore, region confidence Θ_m will be approximately equal to zero or one. Likewise, the strongly model-based region confidence

$$P(E[Q_m] < T) \approx F_{N_m-1} \left(\frac{T - \hat{Q}_m}{\hat{\Psi}_m / \sqrt{N_m}} \right)$$

will be approximately equal to zero or one because the estimate $\hat{\Psi}_m^2$ of region variance is (see Section 4.2)

$$\hat{\Psi}_m^2 = \frac{1}{W_m^2} \sum_{i=1}^{\eta(m)} w_{\kappa(m,i)}^2 \hat{\Sigma}_{\kappa(m,i)}^2 \approx 0.$$

The sign of $T - \hat{Q}_m$ determines whether $P(E[Q_m] < T)$ is approximately equal to zero or one. After a minor rearrangement of terms,

$$T - \hat{Q}_m = \frac{1}{W_m} \sum_{i=1}^{\eta(m)} w_{\kappa(m,i)} (T - \hat{B}_{\kappa(m,i)}).$$

We assumed that $\{T - \hat{B}_{\kappa(m,i)}\}_{i=1}^{\eta(m)}$ have the same sign, so we have shown that $P(E[Q_m] < T) \approx \Theta_m$. The equality is asymptotically exact as all variance estimates $\{\hat{\Sigma}_{\kappa(m,i)}^2\}_{i=1}^{\eta(m)}$ approach zero. This argument applies regardless of the distributions of $\{B_{\kappa(m,i)}\}_{i=1}^{\eta(m)}$, as long as these random variables are mutually independent.

4.4.3 Optimized Regions

We now pursue the definition of optimized regions. Given a slope τ , $0 \leq \tau \leq 1$, define the *gain* of region R_m , $1 \leq m \leq 2^M$, as

$$G(R_m, \tau) = H_m - \tau S_m,$$

where H_m is the region hit and S_m is the region support. Let an *optimized-gain admissible region* R_τ with respect to slope τ , $0 \leq \tau \leq 1$, be an admissible region with the maximum gain $G(R_\tau, \tau)$ over all admissible

regions (this region need not be unique). Optimized-gain admissible regions are easy to define, compute, and analyze, but hard to interpret. Common practice is to define optimized-confidence and optimized-support admissible regions. Admissible region R_* is an *optimized-confidence admissible region* with respect to a given support threshold 1000η , $0 \leq \eta \leq M$, if R_* has the maximum confidence $\Theta_* = H_*/S_*$ among all admissible regions with support of at least 1000η . Likewise, admissible region R_\diamond is an *optimized-support admissible region* with respect to a given confidence threshold θ , $0 \leq \theta \leq 1$, if R_\diamond has the maximum support $S_\diamond = 1000\eta(\diamond)$ among all admissible regions with confidence of at least θ . In other words, we can either fix the region confidence θ and find the largest region R_\diamond with confidence of at least θ , or we can fix the minimum region size (support) 1000η and find the most confident region R_* with support of at least 1000η .

Observe that τ in the definition of an optimized-gain admissible region is the relative importance of support vs. that of confidence. We can find a small region with high confidence or a large region with small confidence, but both objectives cannot be maximized simultaneously. Increasing τ will increase the confidence of the optimized-gain admissible region, but decrease its support. Likewise, decreasing τ will decrease the confidence of the optimized-gain admissible region, but increase its support. Therefore, we can find approximate optimized-confidence and optimized-support admissible regions by a binary search for the value of τ where the respective threshold is barely satisfied. The search can stop at a given level of precision $\Delta\tau$, where the lower bound on $\Delta\tau$ can be found in [52] (they show that the number of steps in this search is logarithmic in the support $1000M$ of the bucket space). This algorithm is approximate because an optimized-confidence (resp. optimized-support) admissible region need not be an optimized-gain admissible region for any value of τ . Yoda et al. [52] argue that this approximation is reasonable for large datasets.

Let us revisit the definition of region ‘goodness’. Geometrically, the buckets with the same value of X are the columns and the buckets with the same value of Y are the rows. An optimized-gain admissible region can be computed in $O(M_X M_Y^2)$ time by a set of rules of the following form. Recall that a region of type W gets wider from left to right (see Figure 4.9). Let $R_W(m, [s, t])$ be the region of type W with maximum gain $f_W(m, [s, t])$ over all admissible regions of type W that end in column m and span rows s through t in this column. Then, either (a) m is the first column of $R_W(m, [s, t])$, or (b) $R_W(m, [s, t])$ includes the region $R_W(m-1, [s', t'])$ with the maximum gain $f_W(m, [s', t'])$ over all admissible regions that end in column $m-1$ and span rows $s' \geq s$ through $t' \leq t$ in this column. [52] keeps the regions with maximum gain for every region type and every triple $(m, [s, t])$ in a dynamic programming table. These locally maximal regions then grow according to a set of rules that compute an optimized-gain admissible region. This efficient greedy algorithm for computing optimized-gain admissible regions depends on the property of the gain function that we refer to as monotonicity. Let $0 \leq \tau \leq 1$ be a slope and $R_{m'}$ and $R_{m''}$ be two admissible regions with gains $G(R_{m'}, \tau) \geq G(R_{m''}, \tau)$. The gain function $G(R_m, \tau)$ is *monotonic* if for any region R_k disjoint with both $R_{m'}$ and $R_{m''}$

$$G(R_{m'} \cup R_k, \tau) \geq G(R_{m''} \cup R_k, \tau),$$

where the union of regions is defined in the obvious way. It is easy to see that our gain function

$$G(R_m, \tau) = H_m - \tau S_m = \sum_{i=1}^{\eta(m)} [1000P(E[B_{\kappa(m,i)}] < T) + 0.5] - 1000\tau\eta(m)$$

is monotonic because it is additive. However, a strongly model-based gain function

$$G^{(M)}(R_m, \tau) = P(E[Q_m] < T) - \tau\eta(m)/M$$

is not monotonic even if we assume independence of bucket random variables $\{B_{\kappa(m,i)}\}_{i=1}^{\eta(m)}$ that make up Q_m . To the best of our knowledge, only monotonic gain functions are known to result in practical algorithms for computing optimized-gain admissible regions.

What happens when no estimates of mean and/or variance are available for some bucket C_k ? The answer to this question depends on problem-specific considerations. As was demonstrated in Section 4.3, it is sometimes possible to provide conservative estimates for these values. For example, we have empirically shown that the expected BEPs of some configurations $\{c_k\}$ are smaller than $T = 10^{-3}$ with confidence $P(E[b_k] < T) \geq 0.995$. Likewise, we know that as the effective SNR approaches negative infinity (in dB), the BEP approaches 0.5, which is the probability of correctly guessing the value of a random bit when the transmitter is turned off. Thus, we can let $P(E[b_k] < T) = 0$ for points with sufficiently small effective SNRs and a reasonable performance threshold T . If no such estimates are available, we can simply omit the missing buckets from the probability computation. This must be done with care because such buckets will contribute nothing to the confidence of the region. This fact can be used to reduce the computational expense of sampling.

This section has highlighted the sometimes contradictory objectives that aggregation must satisfy: permit valid statistical interpretations and afford structure that can be exploited by data mining algorithms. Our approach has been a judicious mix of concepts from both statistics and data mining. We showed that our formulation of the data mining problem lies between the completely model-free approach and the strongly model-based approach. In particular, mutual independence of bucket random variables is crucial to efficient computation of optimized admissible regions. If the covariance terms are non-negligible, our approach is no longer asymptotically equivalent to the strongly model-based approach. Likewise, interactions between the buckets break the monotonicity of the gain function and make one question the greedy region expansion strategy. The next section applies the data mining methodology described here to the example in Section 4.3.

4.5 Optimized-Support Regions for the STTD Example

This section continues the example in Section 4.3. First, we show that optimized-gain regions are both rectilinear and connected for this example. It immediately follows that optimized-support and optimized-confidence regions are also admissible. An optimized-support admissible region is presented next. We show that the elaborate region mining setup leads to simple engineering interpretations. Finally, we look at the performance of data mining when the number of samples is small. Three-fold cross-validation shows that data mining performs well under these circumstances.

4.5.1 Justification of Data Mining for the STTD Example

Let the average SNRs $S_1 = X$ and $S_2 = Y$ partition the space of configurations in Figure 4.2 into disjoint points (buckets) $\{c_k\}_{k=1}^M$, $1 \leq M \leq 1600$. All points have been simulated independently, so the major assumption of the data mining algorithm is satisfied. The true BEP of any configuration is a constant. Thus, the probability that the expected BEP is below any given threshold T is either zero or one, which determines whether or not the corresponding point should be in a ‘good’ region. Without loss of generality, consider only the points with $X \leq Y$, i.e., $S_1 \leq S_2$. It is easy to extend all arguments to $X > Y$, but this adds little to the discussion.

Let c_1 at (x_1, y_1) and c_2 at (x_2, y_1) , $x_1 < x_2 < y_1$, be two points in an optimized-gain region (of arbitrary shape) for some slope $0 < \tau < 1$ (see Figure 4.10). This means that the confidences of these points are one, and thus the expected BEPs of these points are smaller than the performance threshold T . When $x_1, x_2 < y_1$ and y_1 is fixed, the BEP is a monotonically decreasing function of x —increasing x decreases the power imbalance and increases the effective SNR, so the BEP must decrease. Therefore, the expected BEP of any point c_u at (x_u, y_1) , $x_1 < x_u < x_2$, is below the performance threshold T . Thus, the

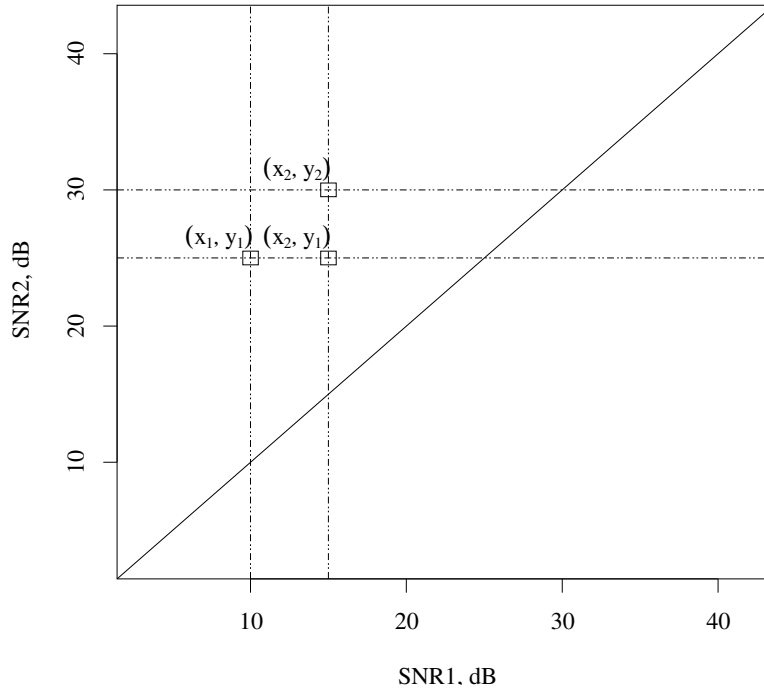


Figure 4.10: Points for arguments about region shape (see text).

confidences of points $\{c_u\}$ are one and these points must be in the optimized-gain region R_τ . Three more symmetric arguments of this kind show that optimized-gain regions are rectilinear.

Likewise, let c_1 at (x_1, y_1) and c_2 at (x_2, y_2) , $x_1 < x_2 < y_1 < y_2$, be two points in an optimized-gain rectilinear region (refer to Figure 4.10). Since c_1 is in the optimized-gain region and $x_1 < x_2$, the point at (x_2, y_1) is also in this region because it has a smaller BEP than c_1 . Since the optimized-gain region is rectilinear, there is a horizontal path from (x_1, y_1) to (x_2, y_1) and a vertical path from (x_2, y_1) to (x_2, y_2) . Thus, there is a Manhattan path from (x_1, y_1) to (x_2, y_2) . Arguments of this kind show that optimized-gain rectilinear regions must be connected as long as they are ‘wide enough’.

To summarize, we have outlined the proof of admissibility of optimized-gain (and thus optimized-support and optimized-confidence) regions. Since the optimized regions are admissible, the data mining algorithm described in Section 4.4 is appropriate for the STTD example. We now show and interpret data mining results.

4.5.2 Optimized-Support Admissible Regions

Figure 4.11 shows an optimized-support admissible region for the confidence threshold $\theta = 0.99$. Intuitively, this is the largest admissible region where we can claim, with confidence of at least 0.99, that configurations exhibit acceptable performance. This claim is conditional on temporal simulation assumptions and on mutual independence of configurations in the region. The shape of this region confirms that, under a fixed effective SNR, the BEP is minimal when the average SNRs of the two branches are equal. The width of this region shows the largest acceptable power imbalance. For this example, the system tolerates power imbalance of up to 12 dB. However, the width of the optimized region is not uniform. The region is narrower for small effective SNRs and wider for large effective SNRs. This means that configurations with

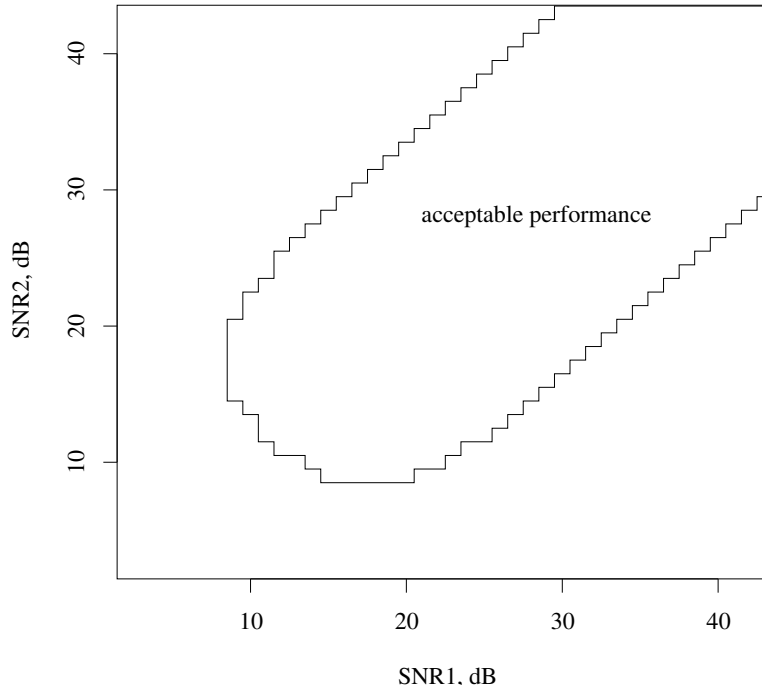


Figure 4.11: Optimized-support admissible region for data in Figure 4.2 (bottom) with the confidence threshold $\theta = 0.99$ and the performance threshold $T = 10^{-3}$.

low effective SNRs are more sensitive to power imbalance than configurations with high effective SNRs. None of these observations are news to an informed reader. The contribution of data mining in this context is not qualitative discoveries; it is statistically significant quantitative results.

Let us see how the data mining algorithm performs when data is scarce. The initial sample of the configuration space in Figure 4.2 (top) contains one sample value per bucket. The statistically significant sample in Figure 4.2 (bottom) contains at least two additional sample values per bucket (recall that we required at least two sample values to estimate bucket variance σ_k^2). Therefore, three-fold cross-validation is the most elaborate cross-validation procedure that this dataset affords. The regions in the top left, bottom left, and top right of Figure 4.12 have been computed for sample values in Figure 4.2 (top) and the first two sample values per bucket in Figure 4.2 (bottom). Each of these regions has been computed with two out of the three sample values per bucket. The region in the lower right has been computed with all data in Figure 4.2 (bottom). All four regions are optimized-support admissible regions with the confidence threshold $\theta = 0.95$. The regions are overlaid on top of the color-coded bucket confidence values. Red (dark) corresponds to low confidence and white (light) corresponds to high confidence that configuration c_k exhibits acceptable average performance w.r.t. the voice quality threshold $T = 10^{-3}$.

The regions in Figure 4.12 are identical except for the lower left corner. This is not surprising because this part of the configuration space exhibits high relative variance. Also, the data is symmetric but the regions are asymmetric in the lower-left corner. Recall that optimized-gain admissible regions, and thus optimized-support admissible regions, are not unique. The ties in region gains are broken arbitrarily. Therefore, region asymmetry is an additional indicator of region instability.

Figure 4.12 also shows that additional data improves image contrast but does not significantly affect region shape. Collecting additional sample values separates the points into ones with low confidence and

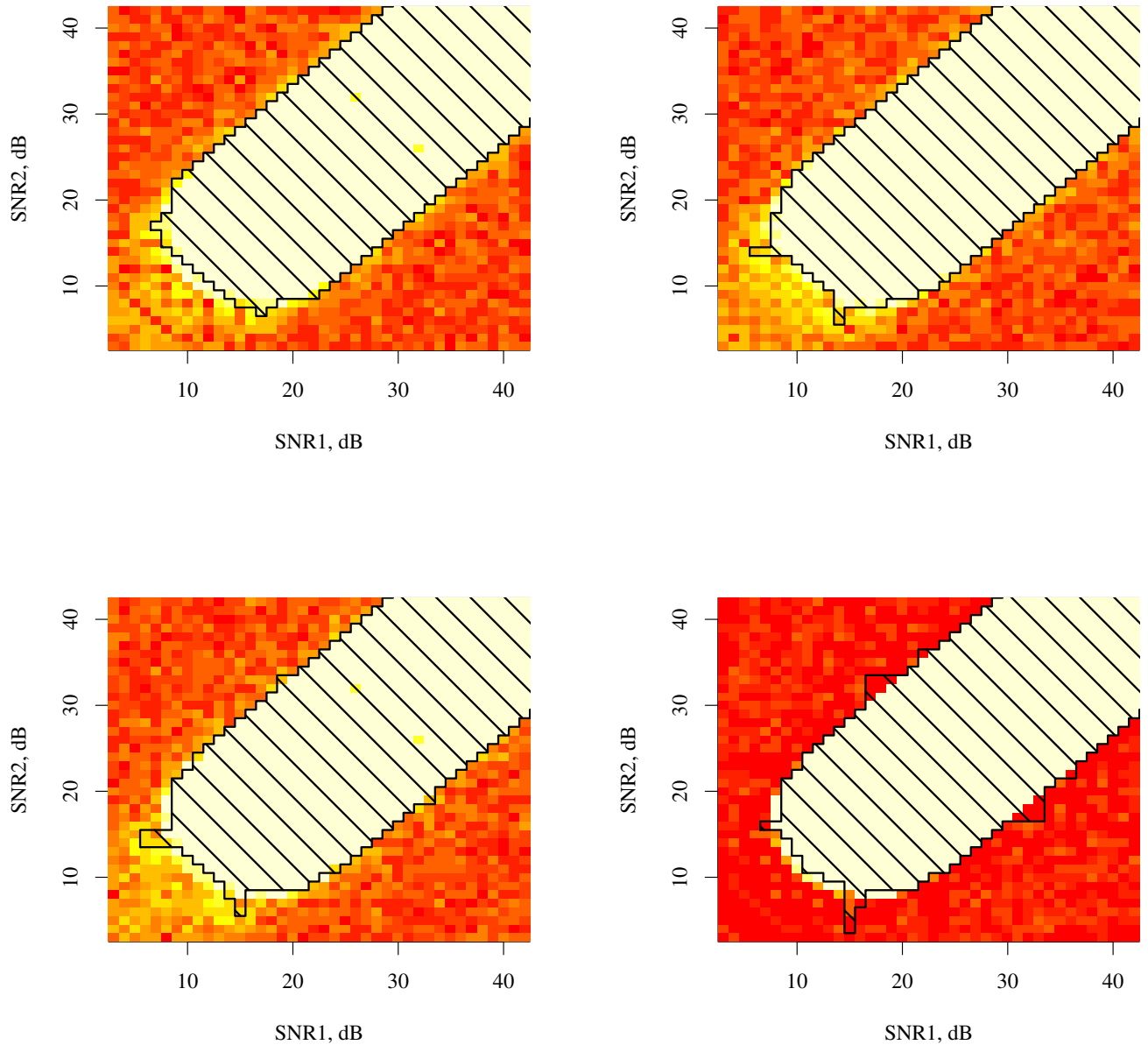


Figure 4.12: Cross-validation of optimized-support admissible regions with the confidence threshold $\theta = 0.95$. The regions in top left, bottom left, and top right have been computed with $n_k = 2$ independent samples per bucket. There are 758 ± 2 buckets (47% of all data) per such region. The region in the bottom right has been computed from the statistically significant data in Figure 4.2 (bottom). It consists of 766 buckets (48% of all data). Red (dark) corresponds to low bucket confidence and white (light) corresponds to high bucket confidence w.r.t. the voice quality threshold $T = 10^{-3}$.

ones with high confidence. A curious side effect occurs when the difference in confidence estimates of low-confidence points falls below the discretization error ($1/1000$). In this case, the ‘confidence slack’ $1 - \theta$ is allocated to arbitrary points with low confidence. One way to correct this situation is to raise the confidence threshold θ —after all, more accurate data should afford stronger claims. Another alternative is to lower the discretization threshold. In general, optimized regions work best when the data is noisy. A contour plot will suffice when the data is highly accurate.

It can also be seen that the high contrast created by the sharp edge of the tolerance region is advantageous to data mining. The region is stable where the contrast is high. When the image is blurred, data mining tries to avoid the questionable boundary points.

To summarize, this section has demonstrated that optimized-gain regions are rectilinear and connected for a non-trivial space of wireless system configurations. We have also shown that optimized-support admissible regions are easy to interpret. Finally, we have shown that data mining works well when sample sizes are small.

4.6 Discussion and Future Work

We have demonstrated a hierarchical formulation of data mining suitable for assessing performance of wireless system configurations. WCDMA simulation results are systematically aggregated and redescribed, leading to intuitive regions that allow the engineer to evaluate wireless system configuration parameters. We have shown that the assumptions about region shape and properties made by data mining algorithms can be valid in the wireless design context; the patterns mined hence lead to explainable and statistically valid design conclusions. As a methodology, data mining is thus shown to be extremely powerful when coupled with statistically meaningful performance evaluation.

This work is the first (known to the authors) application of data mining methodology to solve problems in wireless system design. Therefore, a large number of extensions are possible and called for. We outline possible extensions at the three levels of aggregation: points, buckets, and regions.

At the point level, it may be advantageous to model temporal simulations more precisely. This paper assumes a ‘large enough’ number of frames per simulation and works with the distribution of BEPs. We have shown reasonable analytical and empirical evidence that this distribution is Gaussian. The advantage of this problem formulation is the independence of spatial aggregation from the assumptions of temporal simulation. This helps introduce wireless engineers to the methodology of data mining for studying design problems. However, a stronger model of temporal simulation (e.g., Markov chains in [51]) may yield appreciable gains in software performance. This direction is worth pursuing because few research groups have access to parallel computing facilities of the scale used in this work. For instance, the initial sample of the configuration space in Figure 4.2 (top) would take one year of computation time on a modern workstation. The study presented in this paper would clearly be impossible without significant computational power.

Aggregation of points into buckets is the least developed part of this work. Suppose that we would like to simulate the effects of interference on configuration performance. Assume that the distribution of the average strengths of the interfering signals is known *a priori* (e.g., estimated by ray tracing). We can either make this distribution known to the temporal simulation, or, alternatively, run several temporal simulations for different strengths of interfering signals. The former is more accurate and computationally more efficient, but the latter is more generic and simpler to implement. Bucketing of simulation results with varying simulation parameters is intended to approximate the performance of a single device under varying conditions. This paper does not employ such bucketing but instead builds all the necessary kinds of parameter variation into the temporal simulation (which can be argued to be the right way to do it). However, bucketing may be necessary when one has to work with a given dataset (e.g., measurements). Bucket space can be

viewed as a configuration space for a more complex temporal simulation. Therefore, an in-depth treatment of bucketing is orthogonal to the primary topic of this paper, which is data mining.

Significant work remains to be done at the region level as well. The assumption of mutual independence of buckets will likely remain essential for efficient data mining algorithms. However, the assumption of small variance could conceivably be relaxed. One could also pursue the relatively difficult task of incorporating strongly model-based prior knowledge into the data mining algorithm, or the somewhat easier task of applying different kinds of region mining algorithms to problems in wireless system design.

Defining additional case studies is another obvious direction for future work. We have studied a relatively small part of the parameter space of modern wireless systems. More studies of this type must be performed to highlight the merits and the shortcomings of data mining in this domain.

Finally, the strict staging of data collection and data mining can be relaxed. One can fruitfully interleave the two activities and have the results of data mining drive subsequent data collection. In data-scarce domains, it would be advantageous to focus the data collection effort on only those regions deemed most important to support a particular data mining objective. Methodologies for closing-the-loop in this manner are becoming increasingly prevalent [42]. This will also help define alternative criteria for evaluating experiment designs and layouts.

Bibliography

- [1] N.R. Adam, I. Adiwijaya, T. Critchlow, and R. Musick. Detecting data and schema changes in scientific documents. In *Advances in Digital Libraries*, pages 160–172, 2000.
- [2] V.S. Adve, R. Bagrodia, J.S. Browne, E. Deelman, A. Dube, E.N. Houstis, J.R. Rice, R. Sakellariou, D.J. Sundaram-Stukel, P.J. Teller, and M.K. Vernon. POEMS: End-to-end performance design of large parallel adaptive computational systems. *IEEE Transactions on Software Engineering*, Vol. 26(11):pages 1027–1048, November 2000.
- [3] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.
- [4] A. Aiken, J. Chen, M. Stonebraker, and A. Woodruff. Tioga-2: A direct manipulation database visualization environment. In *Proceedings of the Twelfth International Conference on Data Engineering (ICDE)*, pages 208–217, New Orleans, February 1996.
- [5] S.M. Alamouti. A simple transmit diversity technique for wireless communications. *IEEE Journal on Selected Areas in Communications*, Vol. 16(8):pages 1451–1458, October 1998.
- [6] R.G. Alscher, B.I. Chevone, L.S. Heath, and N. Ramakrishnan. Espresso — a PSE for bioinformatics: Finding answers with microarray technology. In A. Tentner, editor, *Proceedings of the High Performance Computing Symposium, Advanced Simulation Technologies Conference*, pages 64–69, April 2001.
- [7] J.B. Andersen, T.S. Rappaport, and S. Yoshida. Propagation measurements and models for wireless communications channels. *IEEE Communications Magazine*, Vol. 33(1):pages 42–49, January 1995.
- [8] C.R. Anderson. Design and implementation of an ultrabroadband millimeter-wavelength vector sliding correlator channel sounder and in-building multipath measurements at 2.5 & 60 GHz. Master’s thesis, Virginia Tech, May 2002.
- [9] V. Apparao, S. Byrne, M. Champion, S. Isaacs, I. Jacobs, A. Le Hors, G. Nicol, J. Robie, R. Sutor, C. Wilson, and L. Wood. Document object model (DOM) level 1 specification version 1.0. W3C Recommendation Document, October 1998.
- [10] K.K. Bae, J. Jiang, W.H. Tranter, J. He, A. Verstak, L.T. Watson, N. Ramakrishnan, C.R. Anderson, T.S. Rappaport, and C.A. Shaffer. WCDMA STTD performance analysis with transmitter location optimization in indoor systems using ray-tracing technique. In *Proc. IEEE Radio and Wireless Conference (RAWCON)*, Boston, MA, August 2002. To Appear.
- [11] W. Benger, H.-C. Hege, T. Radke, and E. Seidel. Data description via a generalized fiber bundle data model. In *Tenth IEEE International Symposium on High Performance Distributed Computing*, 2001.

- [12] H.P. Bivens. Grid workflow. Grid Computing Environments Working Group Document, Global Grid Forum, 2001.
- [13] R. Bramley, K. Chiu, S. Diwan, D. Gannon, M. Govindaraju, N. Mukhi, B. Temko, and M. Yochuri. A component based services architecture for building distributed applications. In *Proceedings of the Ninth IEEE International Symposium on High Performance Distributed Computing (HPDC'00)*. IEEE Press, 2000.
- [14] F.P. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, anniversary edition, 1995.
- [15] D. Brownell. *SAX2*. O'Reilly Books, January 2002.
- [16] G. Casella and R.L. Berger. *Statistical Inference*. Duxbury, 2nd edition, 2002.
- [17] S. Chawathe and H. Garcia-Molina. Meaningful change detection in structured data. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pages 26–37. Tucson, Arizona, USA, 1997.
- [18] J. Clark and M. Makoto (eds.). RELAX NG specification. OASIS Committee Specification Document, December 2001.
- [19] J. Clark (ed.). XSL transformations (XSLT) version 1.0. W3C Recommendation Document, November 1999.
- [20] T. Critchlow, M. Ganesh, and R. Musick. Meta-data based mediator generation. In *Proceedings of the Third International Conference on Cooperative Information Systems*, pages 168–176, 1998.
- [21] K. Dietze, C.B. Dietrich Jr., and W.L. Stutzman. Analysis of a two-branch maximal ratio and selection diversity system with unequal SNRs and correlated inputs for a Rayleigh fading channel. *IEEE Transactions on Wireless Communications*, Vol. 1(2):pages 274–281, April 2002.
- [22] T.T. Drashansky, E.N. Houstis, N. Ramakrishnan, and J.R. Rice. Networked agents for scientific computing. *Communications of the ACM*, Vol. 42(3):pages 48–54, March 1999.
- [23] U.M. Fayyad, G. Piatetsky-Shapiro, and P. Smyth. The KDD process for extracting useful knowledge from volumes of data. *Communications of the ACM*, Vol. 39(11):pages 27–34, November 1996.
- [24] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.
- [25] B. Freisleben, D. Hartmann, and T. Kielmann. Parallel raytracing: A case study on partitioning and scheduling on workstation clusters. In *Thirtieth International Conference on System Sciences*, volume 1, pages 596–605, Hawaii, 1997.
- [26] J.H. Friedman. Multivariate adaptive regression splines. *Annals of Statistics*, Vol. 19(1):pages 1–141, 1991.
- [27] T. Fukuda, Y. Morimoto, S. Morishita, and T. Tokuyama. Data mining with optimized two-dimensional association rules. *ACM Transactions on Database Systems*, Vol. 26(2):pages 179–213, June 2001.

- [28] E. Gallopoulos, E.N. Houstis, and J.R. Rice. Computer as thinker/doer: Problem-solving environments for computational science. *IEEE Computational Science and Engineering*, Vol. 1(2):pages 11–23, 1994.
- [29] A. Glassner. Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications*, Vol. 4(10):pages 15–22, October 1984.
- [30] A. Goel, C.A. Baker, C.A. Shaffer, B. Grossman, W.H. Mason, L.T. Watson, and R.T. Haftka. VizCraft: A problem-solving environment for aircraft configuration design. *IEEE/AIP Computing in Science and Engineering*, Vol. 3(1):pages 56–66, 2001.
- [31] A. Goel, C. Phanouriou, F.A. Kamke, C.J. Ribbens, C.A. Shaffer, and L.T. Watson. WBCSim: A prototype problem solving environment for wood-based composites simulation. *Engineering with Computers*, Vol. 15:pages 198–210, 1999.
- [32] H. Hashemi. The indoor radio propagation channel. *Proceedings of the IEEE*, Vol. 81(7):pages 943–968, July 1993.
- [33] T. Hastie, R. Tibshirani, and J.H. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer Verlag, October 2001.
- [34] J.M. Hellerstein. Optimization techniques for queries with expensive methods. *ACM Transactions on Database Systems*, Vol. 23(2):pages 113–157, September 1998.
- [35] H. Holma and A. Toskala. *WCDMA for UMTS: Radio Access for Third Generation Mobile Communications*. John Wiley, New York, 2000.
- [36] Y. Ioannidis, M. Livny, S. Gupta, and N. Ponnkanti. ZOO: A desktop experiment management environment. In *Proc. 22nd International VLDB Conference*, pages 274–285, 1996.
- [37] M.C. Jeruchim, P. Balaban, and K.S. Shanmugan. *Simulation of Communication Systems*. Plenum Press, New York, 1992.
- [38] C. Johnson and S. Parker. The SCIRun parallel scientific computing problem solving environment. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, 1999.
- [39] D.R. Jones, C.D. Perttunen, and B.E. Stuckman. Lipschitzian optimization without the Lipschitz constant. *Journal of Optimization Theory and Applications*, Vol. 79(1):pages 157–181, 1993.
- [40] S. Markus, S. Weerawarana, E.N. Houstis, and J.R. Rice. Scientific computing via the world wide web: The net PELLPACK PSE server. *IEEE Computational Science & Engineering*, Vol. 4(3):pages 43–51, July-September 1997.
- [41] S. Pemberton, M. Altheim, D. Austin, F. Boumphrey, J. Burger, A.W. Donoho, S. Dooley, K. Hofrichter, P. Hoschka, M. Ishikawa, W. ten Tate, P. King, P. Klante, S. Matsui, S. McCarron, A. Navarro, Z. Nies, D. Raggett, P. Schmitz, S. Schnitzenbaumer, P. Stark, C. Wilson, T. Wugofski, and D. Zigmond. XHTML 1.0: The extensible hypertext markup language. W3C Recommendation Document, January 2000.
- [42] N. Ramakrishnan and C. Bailey-Kellogg. Sampling strategies for mining in data-scarce domains. *IEEE/AIP Computing in Science and Engineering*, Vol. 4(4):pages 31–43, July/August 2002.

- [43] N. Ramakrishnan and A.Y. Grama. Mining scientific data. *Advances in Computers*, Vol. 55:pages 119–169, September 2001.
- [44] N. Ramakrishnan, E.N. Houstis, and J.R. Rice. Recommender systems for problem solving environments. In H. Kautz, editor, *Technical Report WS-98-08 (Working Notes of the AAAI-98 Workshop on Recommender Systems)*, pages 91–95. AAAI/MIT Press, 1998.
- [45] T.S. Rappaport. *Wireless Communications: Principles and Practice*. Prentice Hall, New Jersey, 1996.
- [46] J.R. Rice and R.F. Boisvert. From scientific software libraries to problem-solving environments. *IEEE Computational Science & Engineering*, Vol. 3(3):pages 44–53, Fall 1996.
- [47] S.Y. Seidel and T.S. Rappaport. Site-specific propagation prediction for wireless in-building personal communication system design. *IEEE Transactions on Vehicular Technology*, Vol. 43(4):pages 879–891, 1994.
- [48] W.L. Stutzman and G.A. Thiele. *Antenna Theory and Design*. John Wiley, New York, second edition, 1998.
- [49] Universal Mobile Telecommunications Systems (UMTS) UMTS Terrestrial Radio Access System (UTRA) Concept Evaluation. Technical Report 101 146 v3.0.0, ETSI, Sophia Antipolis, France, December 1997.
- [50] A. Verstak, J. He, L.T. Watson, N. Ramakrishnan, C.A. Shaffer, T.S. Rappaport, K. Anderson, C.R. Bae, J. Jiang, and W.H. Tranter. S⁴W: Globally optimized design of wireless communication systems. In *Proceedings of the Next Generation Software Workshop, 16th International Parallel and Distributed Processing Symposium (IPDPS'02)*. IEEE Press, April 2002. Fort Lauderdale, FL.
- [51] H.S. Wang. Finite-state Markov channel—a useful model for radio communication channels. *IEEE Transactions on Vehicular Technology*, Vol. 44(1):pages 163–171, February 1995.
- [52] K. Yoda, T. Fukuda, Y. Morimoto, S. Morishita, and T. Tokuyama. Computing optimized rectilinear regions for association rules. In *Proceedings of the Third International Conference on Knowledge Discovery and Data Mining (KDD'97)*, pages 96–103, August 1997.
- [53] R.E. Ziemer and W.H. Tranter. *Principles of Communications: Systems, Modulation, and Noise*. John Wiley, New York, 5th edition, 2002.

Appendix A

EMDAG Schemas

This appendix contains all EMDAG schemas and server-side functions. The schemas and the functions are given in PostgreSQL's dialect of SQL, as of PostgreSQL version 7.2.1. Some functions are written in Tcl. The schemas are well tested and fully functional.

A.1 Installation Management Schemas

```
-- Get the pid of this backend. Backend pid 'more or less' identifies
-- this connection. No two processes can share the same pid, but pids
-- are recycled when processes die. Good news is, it usually takes a
-- while for a pid to get recycled, so this is rarely a problem.
-- result - backend pid
CREATE OR REPLACE FUNCTION backend_pid() RETURNS integer AS '
    return [pid]
' LANGUAGE 'pltclu';

-- Does the backend with a given pid still exist? Keep in mind that
-- process ids are recycled and that this function will also return
-- true for a pid of any other process owned by the Unix user postgres
-- (not necessarily a backend). Therefore, a different process can
-- be mistaken for the backend. However, if this function returns
-- false, you can be sure that the backend does not exist.
-- result - true if the backend 'probably exists', false if definitely not.
CREATE OR REPLACE FUNCTION backend_exists(integer) RETURNS boolean AS '
    if {[string equal $1 ""]} {
        return true
    } elseif {![catch {exec kill -0 $1}]} {
        return true
    } else {
        return false
    }
' LANGUAGE 'pltclu';

-- Component catalog. This catalog lists available components, but
-- does not contain enough information to provide any component
```

```

-- functionality.
-- id - component id
-- description - free-form component description
-- date - component date (when record created)
CREATE TABLE components (
    id          varchar(32)      NOT NULL CHECK (id > ''),
    description text            NOT NULL DEFAULT '',
    date        date            NOT NULL DEFAULT CURRENT_DATE,
    PRIMARY KEY (id)
);

-- Installation catalog.  This catalog lists available installations
-- along with execution management information.
-- id - installation id
-- description - free-form installation description
-- date - installation date (when record created)
-- data_dir - absolute path of data directory (tilde substitution is OK)
-- sh - shell for command execution (Tcl list of arguments)
-- clean_trash_cmd - command to clean stale cache files (Tcl list of arguments)
CREATE TABLE installations (
    id          varchar(32)      NOT NULL CHECK (id > ''),
    description text            NOT NULL DEFAULT '',
    date        date            NOT NULL DEFAULT CURRENT_DATE,
    data_dir    text            NOT NULL DEFAULT '',
    sh          text            NOT NULL DEFAULT '/bin/sh -c',
    clean_trash_cmd text        NOT NULL DEFAULT 'clean_trash -o',
    PRIMARY KEY (id)
);

-- Component installation catalog.  Any component can be installed in
-- multiple places, but you must select a place before running an
-- experiment.  Likewise, any given installation can contain multiple
-- components.
-- component, installation - component installation identification
-- description - free-form installation description
-- date - installation date
-- start_exec_cmd - command to start an executor (Tcl list of arguments)
-- start_exec_arg - reserved for start_exec_cmd (Tcl list of arguments)
-- stop_exec_cmd - command to stop an executor (Tcl list of arguments)
CREATE TABLE component_installations (
    component    varchar(32)      NOT NULL,
    installation  varchar(32)      NOT NULL,
    description   text            NOT NULL DEFAULT '',
    date         date            NOT NULL DEFAULT CURRENT_DATE,
    start_exec_cmd text          NOT NULL DEFAULT 'start_process -o',
    start_exec_arg text          NOT NULL DEFAULT '',
    stop_exec_cmd text          NOT NULL DEFAULT 'stop_process -n',
    PRIMARY KEY (component, installation),
    FOREIGN KEY (component)
        REFERENCES components (id)
);

```

```

        ON DELETE CASCADE,
    FOREIGN KEY (installation)
    REFERENCES installations (id)
    ON DELETE CASCADE
);

-- One can use a remote shell, but it must NOT prompt for a password
-- and it must write ONLY error messages to the standard error stream.
-- A reasonable setting of sh for OpenSSH is
-- '/usr/local/bin/ssh -x -l s4w -o {BatchMode yes} sioux.cs.vt.edu'
-- ('-x' means do not forward X11 connections, '-l s4w' means login as
-- user s4w, '-o {BatchMode yes}' means never prompt for a password, and
-- 'sioux.cs.vt.edu' is host name to connect to). Both sh and commands
-- are treated as Tcl lists of arguments, so Tcl list quoting conventions
-- apply to these values (obviously, SQL quoting conventions also
-- apply to the INSERT statements that create these records).
-- These values are passed to Tcl's exec command like this:
-- 'eval exec $sh $start_exec_command $experiment_oid',
-- 'eval exec $sh $stop_exec_command $executor_id',
-- 'eval exec $sh $clean_trash_cmd $installation_oid',
-- which means that sh itself is not interpolated by the shell, but all
-- commands and their arguments are interpolated by the shell, and all of
-- sh, commands, and arguments are subject to special treatment by Tcl's
-- exec. One may or may not use Tcl exec and shell features in commands,
-- so quoting commands appropriately is left up to the user. Executor ids
-- and oids never contain special characters, so they will not cause any
-- problems (this is why we pass oids, not primary keys). Finally,
-- OpenSSH does not load user's .profile, so remember to set up
-- .ssh/environment appropriately.

-- Is a filename safe to use? A filename is safe to use if it is
-- relative and never refers to any parent directory. Backslashes
-- are not allowed - use forward slashes if you want subdirectories.
-- fname - file or directory name
-- result - true if the filename is safe to use, false if not
CREATE OR REPLACE FUNCTION is_file_safe(text) RETURNS boolean AS '
    # this is somewhat simplistic, but it works...
    if {[string equal [file pathtype $1] relative] &&
        [string first .. $1] < 0 && [string first \\ \\ $1] < 0} {
        return true
    } else {
        return false
    }
' LANGUAGE 'pltclu';

-- Start an executor.
-- id - experiment id
-- result - executor id
CREATE OR REPLACE FUNCTION start_experiment_executor(varchar(32))

```

```

RETURNS integer AS '
spi_exec "SELECT oid, component, installation
        FROM experiments WHERE id=''[quote $1]''
if {[info exists oid]} {
    spi_exec "SELECT sh, start_exec_cmd
            FROM installations i, component_installations ci
            WHERE i.id=''[quote $installation]'' AND
                  ci.component=''[quote $component]'' AND
                  ci.installation=''[quote $installation]''
    if {[info exists start_exec_cmd]} {
        if {[catch {
            set executor_id [eval exec $sh $start_exec_cmd $oid]
        } msg]} {
            elog ERROR "start_executor: $msg"
            return -1
        }
        return $executor_id
    } else {
        elog ERROR "start_executor: no start executor command for \\
                component $component, $installation"
        return -1
    }
} else {
    elog ERROR "start_executor: experiment $1 does not exist"
    return -1
}
' LANGUAGE 'pltclu';

```

```

-- Stop an executor.  This function does *not* abort if the executor
-- does not exist or could not be stopped.
-- id - executor id
-- result - true if the executor was stopped, false if an error occurred
CREATE OR REPLACE FUNCTION stop_experiment_executor(integer)

```

```

RETURNS boolean AS '
spi_exec "LOCK TABLE experiment_executors IN SHARE ROW EXCLUSIVE MODE"
spi_exec "SELECT exp_id FROM experiment_executors WHERE id=$1"
if {[info exists exp_id]} {
    spi_exec "SELECT sh, stop_exec_cmd
            FROM experiments e, installations i,
            component_installations ci
            WHERE e.id=''[quote $exp_id]'' AND
                  i.id=e.installation AND
                  ci.component=e.component AND
                  ci.installation=e.installation"
    if {[info exists stop_exec_cmd]} {
        set r true
        if {[catch {
            eval exec $sh $stop_exec_cmd $1
        } msg]} {
            elog NOTICE "stop_executor: $msg"
            set r false
        }
    }
}

```

```

    }
    # we are in charge of cleaning up
    spi_exec "SELECT unregister_experiment_executor($1)"
    return $r
  } else {
    # this is a serious problem
    elog ERROR "stop_executor: no stop executor command for \\
              experiment $exp_id"
    return false
  }
} else {
  elog NOTICE "stop_executor: executor $1 does not exist"
  return false
}
' LANGUAGE 'pltclu';

```

```

-- Clean trash files for an installation. This function does *not*
-- abort if the files could not be cleaned.
-- installation - installation to clean
CREATE OR REPLACE FUNCTION clean_trash(varchar(32)) RETURNS boolean AS '
  spi_exec "SELECT oid, sh, clean_trash_cmd
            FROM installations i
            WHERE i.id=''[quote $1]''"
  spi_exec "SELECT COUNT(*) AS count FROM installation_trash
            WHERE installation=''[quote $1]''"
  if {$count <= 0} {
    return true
  }
  if {[info exists clean_trash_cmd]} {
    if {[catch {
      eval exec $sh $clean_trash_cmd $oid
    } msg]} {
      elog NOTICE "clean_trash: $msg"
      return false
    }
    # delete records of trash
    spi_exec "DELETE FROM installation_trash
              WHERE installation=''[quote $1]''"
    return true
  } else {
    elog NOTICE "clean_trash: installation $1 not found"
    return false
  }
}
' LANGUAGE 'pltclu';

```

```

-- Large object cache. Any component can cache a number of large
-- objects in its local filesystem. These cache files are shared by
-- all components in the installation. When records are deleted from
-- this table, the filenames of the cache files are automatically
-- inserted in the trash table. These stale files should be

```

```

-- cleaned up once in a while.  Cached files are not automatically
-- deleted if the component installation is removed.
-- installation - installation id
-- loid - large object id
-- fname - relative name of the cache file (to installation's data directory)
CREATE TABLE installation_cache (
    installation    varchar(32)    NOT NULL,
    loid           oid            NOT NULL,
    fname          text           NOT NULL,
    CHECK (fname > '' AND is_file_safe(fname)),
    PRIMARY KEY (installation, loid),
    FOREIGN KEY (installation)
        REFERENCES installations (id)
        ON DELETE CASCADE
);
CREATE OR REPLACE FUNCTION cache_to_trash() RETURNS OPAQUE AS '
    spi_exec \\
        "INSERT INTO installation_trash (installation, fname)
        VALUES (''[quote $OLD(installation)]'', ''[quote $OLD(fname)]'')"
    return OK
' LANGUAGE 'pltcl';
CREATE TRIGGER cache_to_trash
    AFTER DELETE ON installation_cache
    FOR EACH ROW EXECUTE PROCEDURE cache_to_trash();

-- Trash bin for invalid cache files.  This table contains the
-- filenames of cache files that are no longer used.  Such files
-- should be periodically cleaned up.  Trash files are not automatically
-- deleted if the component installation is removed.
-- installation - installation identification
-- fname - relative name of the trash file (to installation's data directory)
CREATE TABLE installation_trash (
    installation    varchar(32)    NOT NULL,
    fname          text           NOT NULL,
    CHECK (fname > '' AND is_file_safe(fname)),
    FOREIGN KEY (installation)
        REFERENCES installations (id)
        ON DELETE CASCADE
);
CREATE INDEX installation_trash_index
    ON installation_trash (installation);

```

A.2 Execution Management Schemas

```

-- Experiment catalog.
-- id - experiment id
-- component, installation - component to execute
-- description - free-form experiment description
-- date - experiment date (when created)
-- owner - experiment owner (Postgres user name)

```

```

-- np - number of processors (if applicable)
-- debug - debug level: -1 (quiet), 0 (normal), 1 (verbose), ...
-- num_controllers - total number of controllers ever successfully started
-- num_executors - total number of executors ever successfully started
-- Example:
-- SELECT id, description, date, np FROM experiments
-- ORDER BY date DESC
CREATE TABLE experiments (
    id          varchar(32)      NOT NULL CHECK (id > ''),
    component   varchar(32)      NOT NULL,
    installation varchar(32)      NOT NULL DEFAULT 'default',
    description text             NOT NULL DEFAULT '',
    date        date             NOT NULL DEFAULT CURRENT_DATE,
    owner       varchar(32)      NOT NULL DEFAULT CURRENT_USER,
    np          integer          NOT NULL DEFAULT '1' CHECK (np >= 1),
    debug       integer          NOT NULL DEFAULT '0',
    num_controllers integer      NOT NULL DEFAULT '0',
    num_executors integer        NOT NULL DEFAULT '0',
    PRIMARY KEY (id),
    FOREIGN KEY (component, installation)
        REFERENCES component_installations (component, installation)
        ON DELETE RESTRICT
);

-- Only superuser can create experiments not owned by them.
CREATE OR REPLACE FUNCTION experiment_insert_trigger() RETURNS OPAQUE AS '
    # constraints may not have been checked yet
    if {![info exists NEW(owner)]} {
        return OK
    }
    spi_exec "SELECT (CURRENT_USER=''[quote $NEW(owner)]'' OR usesuper) AS ok
        FROM pg_user WHERE username=CURRENT_USER"
    if {![string match -nocase t* $ok]} {
        elog ERROR "experiment_insert_trigger: must be $NEW(owner) \\
            or superuser to create experiments owned by $NEW(owner)"
    }
    return OK
' LANGUAGE 'pltcl';
CREATE TRIGGER experiment_insert_trigger
    BEFORE INSERT
    ON experiments
    FOR EACH ROW EXECUTE PROCEDURE experiment_insert_trigger();

-- Only superuser can change experiment owners. However, anyone who
-- has permissions to update the experiments table can change other
-- experiment attributes, such as descriptions.
CREATE OR REPLACE FUNCTION experiment_update_trigger() RETURNS OPAQUE AS '
    # constraints may not have been checked yet
    if {![info exists NEW(owner)]} {
        return OK
    }

```

```

    }
    if {![string equal $OLD(owner) $NEW(owner)]} {
        spi_exec "SELECT usesuper AS ok FROM pg_user WHERE username=CURRENT_USER"
        if {![info exists ok]} {
            set ok false
        }
        if {![string match -nocase t* $ok]} {
            elog ERROR "experiment_update_trigger: must be superuser \\
                to change experiment owner"
        }
    }
}
return OK
' LANGUAGE 'pltcl';
CREATE TRIGGER experiment_update_trigger
    BEFORE UPDATE
    ON experiments
    FOR EACH ROW EXECUTE PROCEDURE experiment_update_trigger();

-- Only owner or superuser can delete an experiment. Also, stop executors
-- (but not controllers) before an experiment is deleted.
CREATE OR REPLACE FUNCTION experiment_delete_trigger() RETURNS OPAQUE AS '
    spi_exec "SELECT (CURRENT_USER=''[quote $OLD(owner)]'' OR usesuper) AS ok
        FROM pg_user WHERE username=CURRENT_USER"
    if {![info exists ok]} {
        set ok false
    }
    if {![string match -nocase t* $ok]} {
        elog ERROR "experiment_delete_trigger: must be $OLD(owner) \\
            or superuser to delete experiment"
    }
    spi_exec "SELECT stop_experiment_executor(id)
        FROM experiment_executors WHERE exp_id=''[quote $OLD(id)]''"
    return OK
' LANGUAGE 'pltcl';
CREATE TRIGGER experiment_delete_trigger
    BEFORE DELETE
    ON experiments
    FOR EACH ROW EXECUTE PROCEDURE experiment_delete_trigger();

-- Send experiment notifications. Note that the events will be
-- replayed during batch data loads, which can make things quite slow.
-- (Triggers are invoked on COPY FROM, so this is not a solution.)
CREATE OR REPLACE FUNCTION experiment_events_trigger() RETURNS OPAQUE AS '
    switch -exact $TG_op {
        INSERT {
            spi_exec "NOTIFY experiments_insert"
        }
        UPDATE {
            spi_exec "NOTIFY experiments_update"
        }
    }
'

```

```

        DELETE {
            spi_exec "NOTIFY experiments_delete"
        }
    }
    return OK
' LANGUAGE 'pltcl';
CREATE TRIGGER experiment_events_trigger
    AFTER INSERT OR UPDATE OR DELETE
    ON experiments
    FOR EACH ROW EXECUTE PROCEDURE experiment_events_trigger();

-- Experiment point computation states.  Each point starts out fresh,
-- then spends some time being computed, after which it becomes computed
-- or crashed.  Users can reset computed or crashed points back to the
-- fresh state.  Points are not marked as deleted, but are permanently
-- deleted from the database by the DELETE command.
-- id, seq_no - identifies experiment point
-- state - 0 (fresh), 1 (executing), 2 (computed), or 3 (crashed)
CREATE TABLE experiment_point_states (
    id            varchar(32)    NOT NULL,
    seq_no        integer        NOT NULL,
    state         integer        NOT NULL DEFAULT '0',
    CHECK (state >= 0 AND state <= 3),
    PRIMARY KEY (id, seq_no),
    FOREIGN KEY (id) REFERENCES experiments (id)
        ON DELETE CASCADE
);

-- Reset crashed points to fresh state.
-- id - experiment id
-- result - number of points reset, zero if no such experiment
CREATE OR REPLACE FUNCTION reset_crashed_points(varchar(32))
    RETURNS integer AS '
    spi_exec "SELECT clean_experiment_executors()"
    spi_exec "DELETE FROM experiment_point_diagnostics WHERE EXISTS (
        SELECT * FROM experiment_point_states s
        WHERE s.id = experiment_point_diagnostics.id AND
              s.seq_no = experiment_point_diagnostics.seq_no AND
              s.state = 3 AND s.id = '[quote $1]'"
    return [spi_exec "UPDATE experiment_point_states SET state = 0
        WHERE state = 3 AND id = '[quote $1]'"
' LANGUAGE 'pltcl';

-- Reset all (computed or crashed, but not computing) points to fresh state.
-- id - experiment id
-- result - number of points reset, zero if no such experiment
CREATE OR REPLACE FUNCTION reset_all_points(varchar(32)) RETURNS integer AS '
    spi_exec "SELECT clean_experiment_executors()"
    spi_exec "DELETE FROM experiment_point_diagnostics WHERE EXISTS (

```

```

        SELECT * FROM experiment_point_states s
        WHERE s.id = experiment_point_diagnostics.id AND
              s.seq_no = experiment_point_diagnostics.seq_no AND
              s.state > 1 AND s.id = '[quote $1]''
    return [spi_exec "UPDATE experiment_point_states SET state = 0
                    WHERE state > 1 AND id = '[quote $1]''"]
' LANGUAGE 'pltcl';

-- Automatically initialize experiment point state to zero.
-- Use this function as a 'BEFORE INSERT' or 'BEFORE UPDATE' trigger
-- on experiment input tables that reference experiment_point_states
-- via foreign keys. Such setup obviates the need to insert fresh
-- experiment point records into the experiment_point_states table.
-- This function will insert such records automatically.
-- id - name of the experiment id column
-- seq_no - name of the experiment sequence number column
CREATE OR REPLACE FUNCTION auto_init_point_state() RETURNS OPAQUE AS '
    if {[info exists NEW($1)] && [info exists NEW($2)]} {
        set n 0
        spi_exec "SELECT COUNT(*) AS n FROM experiment_point_states
                WHERE id = '[quote $NEW($1)]' AND seq_no = $NEW($2)"
        if {$n <= 0} {
            spi_exec "INSERT INTO experiment_point_states (id, seq_no, state)
                    VALUES ('[quote $NEW($1)]', $NEW($2), 0)"
        }
    }
    return [array get NEW]
' LANGUAGE 'pltcl';

-- Get condition variable name for 'loaded fresh point' notifications.
-- These notifications are sent when experiment point inputs are loaded
-- into the experiment_point_states table.
-- id - experiment id (one condition variable is allocated per experiment)
-- result - condition variable name, or NULL if no such experiment
CREATE OR REPLACE FUNCTION experiment_point_fresh_cond(varchar(32))
    RETURNS text AS '
    SELECT 'experiment_' || oid || '_fresh' FROM experiments WHERE id = $1;
' LANGUAGE 'sql';

-- Get condition variable name for 'start computing point' notifications.
-- These notifications are sent when experiment points are marked computing
-- in the experiment_point_states table.
-- id - experiment id (one condition variable is allocated per experiment)
-- result - condition variable name, or NULL if no such experiment
CREATE OR REPLACE FUNCTION experiment_point_start_cond(varchar(32))
    RETURNS text AS '
    SELECT 'experiment_' || oid || '_start' FROM experiments WHERE id = $1;
' LANGUAGE 'sql';

```

```

-- Get condition variable name for 'done computing point' notifications.
-- These notifications are sent when experiment points are marked computed
-- or crashed in the experiment_point_states table.
-- id - experiment id (one condition variable is allocated per experiment)
-- result - condition variable name, or NULL if no such experiment
CREATE OR REPLACE FUNCTION experiment_point_done_cond(varchar(32))
  RETURNS text AS '
  SELECT 'experiment_' || oid || '_done' FROM experiments WHERE id = $1;
' LANGUAGE 'sql';

-- Get condition variable name for 'point deleted' notifications.
-- These notifications are sent when experiment points are deleted from
-- the experiment_point_states table.
-- id - experiment id (one condition variable is allocated per experiment)
-- result - condition variable name, or NULL if no such experiment
CREATE OR REPLACE FUNCTION experiment_point_deleted_cond(varchar(32))
  RETURNS text AS '
  SELECT 'experiment_' || oid || '_deleted'
  FROM experiments WHERE id = $1;
' LANGUAGE 'sql';

-- Send experiment point notifications. Note that the events will be
-- replayed during batch data loads, which can make things quite slow.
-- (Triggers are invoked on COPY FROM, so this is not a solution.)
CREATE OR REPLACE FUNCTION experiment_point_events_trigger()
  RETURNS OPAQUE AS '
  if {[info exists NEW(id)]} {
    set qid '['quote $NEW(id)']'
    set state $NEW(state)
  } elseif {[info exists OLD(id)]} {
    set qid '['quote $OLD(id)']'
    set state $OLD(state)
  } else {
    # this is probably an error, but who am I to judge
    return OK
  }
  if {[spi_exec "SELECT experiment_point_fresh_cond($qid) AS fresh_cond,
                experiment_point_start_cond($qid) AS start_cond,
                experiment_point_done_cond($qid) AS done_cond,
                experiment_point_deleted_cond($qid) AS deleted_cond"] != 1} {
    # likewise
    return OK
  }
  if {![info exists fresh_cond]} {
    # likewise
    return OK
  }
  switch -exact $TG_op {
    INSERT {

```

```

        # replay events on insert
        if {$state >= 0} { spi_exec "NOTIFY $fresh_cond" }
        if {$state >= 1} { spi_exec "NOTIFY $start_cond" }
        if {$state >= 2} { spi_exec "NOTIFY $done_cond" }
    }
    UPDATE {
        # only notify for common types of updates
        if {[string equal $NEW(id) $OLD(id)]} {
            if {$OLD(state) < $state} {
                # state was advanced
                if {$OLD(state) == 0} {
                    spi_exec "NOTIFY $start_cond"
                    if {$state >= 2} { spi_exec "NOTIFY $done_cond" }
                } elseif {$OLD(state) == 1} {
                    spi_exec "NOTIFY $done_cond"
                }
            } elseif {$OLD(state) > 0 && $state == 0} {
                # state was reset (e.g., after crash)
                spi_exec "NOTIFY $fresh_cond"
            }
        }
    }
}
DELETE {
    spi_exec "NOTIFY $deleted_cond"
}
}
return OK
' LANGUAGE 'pltcl';
CREATE TRIGGER experiment_point_events_trigger
    AFTER INSERT OR UPDATE OR DELETE
    ON experiment_point_states
    FOR EACH ROW EXECUTE PROCEDURE experiment_point_events_trigger();

-- Experiment point diagnostic messages.  This table is populated by
-- experiment executors, i.e., diagnostic messages accumulate in the
-- background as the experiments are executing.
-- id, seq_no - experiment point identification
-- diag_no - identifies diagnostic message and enforces ordering
-- diag_type - 0 (debug), 1 (warning), or 2 (error)
-- diag_msg - free-form diagnostic message
CREATE SEQUENCE ept_diagnostic_ids;
CREATE TABLE experiment_point_diagnostics (
    id          varchar(32)    NOT NULL,
    seq_no     integer        NOT NULL,
    diag_no    integer        NOT NULL
                                DEFAULT nextval('ept_diagnostic_ids'),
    diag_type  integer        NOT NULL DEFAULT '0',
    diag_msg   text           NOT NULL DEFAULT 'got here',
    PRIMARY KEY (diag_no),
    FOREIGN KEY (id, seq_no)
    REFERENCES experiment_point_states (id, seq_no)

```

```

        ON DELETE CASCADE
    );

-- Concatenate two strings and put a newline in-between.
-- s1, s2 - strings to concatenate
-- result - s1 || '\n' || s2
CREATE OR REPLACE FUNCTION msg_concat(text, text) RETURNS text AS '
    SELECT $1 || '\n' || $2;
' LANGUAGE 'sql';

-- Concatenate a string and a newline.
-- s - string
-- result - s || '\n', or empty if s was empty
CREATE OR REPLACE FUNCTION msg_concat(text) RETURNS text AS '
    SELECT CASE WHEN $1 > '' THEN $1 || '\n'
              ELSE ''
    END;
' LANGUAGE 'sql';

-- Concatenate message type and contents.
-- type - message type: 0 (debug), 1 (warning), or 2 (error)
-- msg - message contents
-- result - message with type string prepended
CREATE OR REPLACE FUNCTION msg_label(integer, text) RETURNS text AS '
    SELECT CASE WHEN $1 = 0 THEN 'debug: ' || $2
              WHEN $1 = 1 THEN 'warning: ' || $2
              WHEN $1 = 2 THEN 'error: ' || $2
              ELSE 'unknown: ' || $2
    END;
' LANGUAGE 'sql';

-- Concatenate a bunch of messages.
CREATE AGGREGATE msg_concat (
    INITCOND = '',
    BASETYPE = text,
    STYPE = text,
    SFUNC = msg_concat,
    FINALFUNC = msg_concat
);

-- Summary of experiment point diagnostic messages.
-- id, seq_no - experiment point identification
-- messages - concatenated diagnostic messages, or empty string if none
CREATE VIEW ept_diagnostics_summary AS
    SELECT
        d.id AS id,
        d.seq_no AS seq_no,

```

```

        MSG_CONCAT(msg_label(d.diag_type,d.diag_msg)) AS messages
FROM experiment_point_diagnostics d
GROUP BY id, seq_no;

-- Experiment executors that are currently running.  There can be
-- zero or more executors running at any time for any experiment.
-- Each executor may (or may not) be a controller for a different
-- experiment.  At most one executor can be computing any given
-- point at the same time.
-- id - executor id
-- pid - executor connection id (backend pid), or NULL if job is queued
-- exp_id - experiment id
-- exp_seq_no - experiment point seq_no, or NULL initially
-- job_id - executor's job id (assigned by batch system), or NULL if unknown
-- state - 0 (queued), 1 (idle), or 2 (running)
CREATE SEQUENCE experiment_executor_ids;
CREATE TABLE experiment_executors (
    id            integer            NOT NULL
                DEFAULT nextval('experiment_executor_ids'),
    pid           integer,
    exp_id        varchar(32)       NOT NULL,
    exp_seq_no    integer,
    job_id        varchar(32),
    state         integer            NOT NULL DEFAULT '0',
    PRIMARY KEY (id),
    FOREIGN KEY (exp_id)
    REFERENCES experiments (id)
        ON DELETE RESTRICT,
    FOREIGN KEY (exp_id, exp_seq_no)
    REFERENCES experiment_point_states (id, seq_no)
        ON DELETE RESTRICT,
    UNIQUE (exp_id, exp_seq_no)
);

-- Get condition variable name for 'executor initialized' notifications.
-- These notifications are sent when executors are inserted into
-- the experiment_executors table.
-- id - experiment id (one condition variable is allocated per experiment)
-- result - condition variable name, or NULL if no such experiment
CREATE OR REPLACE FUNCTION experiment_executor_init_cond(varchar(32))
    RETURNS text AS '
    SELECT 'experiment_' || oid || '_exec_init' FROM experiments
    WHERE id = $1;
' LANGUAGE 'sql';

-- Get condition variable name for 'executor started' notifications.
-- These notifications are sent when queued executors start up,
-- i.e., upon executor registration.
-- id - experiment id (one condition variable is allocated per experiment)

```

```

-- result - condition variable name, or NULL if no such experiment
CREATE OR REPLACE FUNCTION experiment_executor_start_cond(vvarchar(32))
  RETURNS text AS '
  SELECT 'experiment_' || oid || '_exec_start' FROM experiments
  WHERE id = $1;
' LANGUAGE 'sql';

-- Get condition variable name for 'executor stopped' notifications.
-- These notifications are sent when executors are deleted from
-- the experiment_executors table.
-- id - experiment id (one condition variable is allocated per experiment)
-- result - condition variable name, or NULL if no such experiment
CREATE OR REPLACE FUNCTION experiment_executor_stop_cond(vvarchar(32))
  RETURNS text AS '
  SELECT 'experiment_' || oid || '_exec_stop' FROM experiments
  WHERE id = $1;
' LANGUAGE 'sql';

-- Send experiment executor notifications. Experiment executor table
-- should never be batch-loaded, so don't assume any specific batch
-- loading event semantics.
CREATE OR REPLACE FUNCTION experiment_executors_trigger() RETURNS OPAQUE AS '
  if {[info exists NEW(exp_id)]} {
    set qid "[quote $NEW(exp_id)]"
  } elseif {[info exists OLD(exp_id)]} {
    set qid "[quote $OLD(exp_id)]"
  } else {
    # this is probably an error, but who am I to judge
    return OK
  }
  if {[spi_exec \
    "SELECT experiment_executor_init_cond($qid) AS init_cond,
    experiment_executor_stop_cond($qid) AS stop_cond"] != 1} {
    # likewise
    return OK
  }
  if {[!info exists init_cond]} {
    # likewise
    return OK
  }
  switch -exact $TG_op {
    INSERT {
      spi_exec "UPDATE experiments SET num_executors = num_executors+1
      WHERE id=$qid"
      spi_exec "NOTIFY $init_cond"
    }
    DELETE {
      spi_exec "NOTIFY $stop_cond"
    }
  }
}

```

```

    return OK
' LANGUAGE 'pltcl';
CREATE TRIGGER experiment_executors_trigger
    AFTER INSERT OR DELETE
    ON experiment_executors
    FOR EACH ROW EXECUTE PROCEDURE experiment_executors_trigger();

-- Register an experiment executor. This function must be called by
-- the executor before requesting any points.
-- id - executor id
CREATE OR REPLACE FUNCTION register_experiment_executor(integer)
    RETURNS integer AS '
    # this is crude, but very effective
    spi_exec "LOCK TABLE experiment_point_states IN SHARE ROW EXCLUSIVE MODE"
    spi_exec "LOCK TABLE experiment_executors IN SHARE ROW EXCLUSIVE MODE"
    # make sure the executor exists
    if {[spi_exec "SELECT exp_id, state \\  

        FROM experiment_executors WHERE id=$1"] != 1} {
        elog ERROR "register_experiment_executor: executor $1 does not exist"
        return_null
    }
    if {$state != 0} {
        elog ERROR "register_experiment_executor: called out of order, \\  

            executor state=$state"
        return_null
    }
    # the executor is now idle and its backend pid is known
    spi_exec "UPDATE experiment_executors SET pid=backend_pid(), state=1
        WHERE id=$1"
    # broadcast registration event
    spi_exec "SELECT experiment_executor_start_cond(exp_id) AS start_cond
        FROM experiment_executors WHERE id=$1"
    spi_exec "NOTIFY $start_cond"
    # ignore return value
    return 1
' LANGUAGE 'pltcl';

-- Grab an experiment point to compute.
-- id - executor id
-- result - point seq_no if non-negative, -1 if experiment is complete,
--          or NULL to wait for a notification and try again
CREATE OR REPLACE FUNCTION grab_experiment_point(integer) RETURNS integer AS '
    # this is crude, but very effective
    spi_exec "LOCK TABLE experiment_point_states IN SHARE ROW EXCLUSIVE MODE"
    spi_exec "LOCK TABLE experiment_executors IN SHARE ROW EXCLUSIVE MODE"
    # find experiment id
    if {[spi_exec "SELECT exp_id, state FROM experiment_executors
        WHERE id=$1"] != 1} {
        elog ERROR "grab_experiment_point: executor $1 does not exist"
        return_null
    }

```

```

    }
    if {$state != 1} {
        elog ERROR "grab_experiment_point: called out of order, \\\
                    executor state=$state"

        return_null
    }
    # clean up a little
    spi_exec "SELECT clean_experiment_executors()"
    # grab the first uncomputed point
    spi_exec "SELECT MIN(s.seq_no) AS seq_no FROM experiment_point_states s
              WHERE s.id=''[quote $exp_id]'' AND s.seq_no >= 0 AND state=0"
    if {[info exists seq_no]} {
        # the executor is now running and computing this point
        spi_exec "UPDATE experiment_point_states SET state=1
                  WHERE id=''[quote $exp_id]'' AND seq_no=$seq_no"
        spi_exec "UPDATE experiment_executors SET exp_seq_no=$seq_no, state=2
                  WHERE id=$1"
        return $seq_no
    } else {
        # could not grab any point, see why not
        spi_exec "SELECT MIN(c.id) AS ctrl FROM experiment_controllers c
                  WHERE c.exp_id=''[quote $exp_id]'' AND
                  backend_exists(c.pid)"
        if {[info exists ctrl]} {
            # at least one controller exists => wait for a while
            return_null
        } else {
            # no controllers => assume the experiment is complete
            return -1
        }
    }
}
' LANGUAGE 'pltcl';

-- Release the experiment point being computed.
-- id - executor id
-- state - 2 (computed) or 3 (crashed)
CREATE OR REPLACE FUNCTION release_experiment_point(integer, integer)
RETURNS integer AS '
# this is crude, but very effective
spi_exec "LOCK TABLE experiment_point_states IN SHARE ROW EXCLUSIVE MODE"
spi_exec "LOCK TABLE experiment_executors IN SHARE ROW EXCLUSIVE MODE"
# find experiment id
if {$2 != 2 && $2 != 3} {
    elog ERROR "release_experiment_point: invalid new point state: $2"
    return_null
}
if {[spi_exec "SELECT exp_id, exp_seq_no, state
              FROM experiment_executors WHERE id=$1"] != 1} {
    elog ERROR "release_experiment_point: executor $1 does not exist"
    return_null
}
}

```

```

if {$state != 2} {
    elog ERROR "release_experiment_point: called out of order, \\  

                executor state=$state"
    return_null
}
# executor is now idle
spi_exec "UPDATE experiment_executors SET state=1 WHERE id=$1"
# mark point complete or crashed
if {[spi_exec "UPDATE experiment_point_states SET state=$2
              WHERE id=''[quote $exp_id]'' AND seq_no=$exp_seq_no AND
              state=1"] != 1} {
    elog NOTICE "release_experiment_point: point ($exp_id, $exp_seq_no) \\  

                was apparently deleted"
}
# ignore return value
return 1
' LANGUAGE 'pltcl';

-- Unregister an experiment executor. The executor must not request or
-- compute any points after this function has been called.
-- id - executor id
CREATE OR REPLACE FUNCTION unregister_experiment_executor(integer)
RETURNS integer AS '
# this is crude, but very effective
spi_exec "LOCK TABLE experiment_point_states IN SHARE ROW EXCLUSIVE MODE"
spi_exec "LOCK TABLE experiment_executors IN SHARE ROW EXCLUSIVE MODE"
# make sure the executor exists
if {[spi_exec "SELECT exp_id, exp_seq_no, state
              FROM experiment_executors
              WHERE id=$1"] != 1} {
    elog ERROR "unregister_experiment_executor: executor $1 does not exist"
    return_null
}
# if it was computing a point, assume the simulation crashed
if {$state == 2} {
    if {[spi_exec "UPDATE experiment_point_states SET state=3
                  WHERE id=''[quote $exp_id]'' AND
                  seq_no=$exp_seq_no"] == 1} {
        spi_exec "INSERT INTO experiment_point_diagnostics (
                  id, seq_no, diag_type, diag_msg
                ) VALUES (
                  ''[quote $exp_id]'', $exp_seq_no, 2,
                  ''executor apparently crashed'' )"
    }
}
# delete this executor
spi_exec "DELETE FROM experiment_executors WHERE id=$1"
# ignore return value
return 1
' LANGUAGE 'pltcl';

```

```

-- Clean stale executor records.
-- result - number of stale executor records cleaned
CREATE OR REPLACE FUNCTION clean_experiment_executors() RETURNS integer AS '
    SELECT SUM(unregister_experiment_executor(id))::integer AS result
    FROM experiment_executors WHERE NOT backend_exists(pid);
' LANGUAGE 'sql';

-- Experiment controllers that are currently running. There can be
-- zero or more controllers running at any time for any experiment.
-- id - controller id
-- exec_id - optional executor id if this controller doubles up as an executor
-- pid - controller connection id (backend pid), or NULL if job is queued
-- exp_id - experiment id
CREATE SEQUENCE experiment_controller_ids;
CREATE TABLE experiment_controllers (
    id                integer          NOT NULL
                        DEFAULT nextval('experiment_controller_ids'),
    exec_id           integer,
    pid               integer,
    exp_id            varchar(32)      NOT NULL,
    PRIMARY KEY (id),
    FOREIGN KEY (exec_id) REFERENCES experiment_executors (id)
        ON DELETE CASCADE,
    FOREIGN KEY (exp_id) REFERENCES experiments (id)
        ON DELETE RESTRICT
);

-- Get condition variable name for 'controller initialized' notifications.
-- These notifications are sent when controllers are inserted into
-- the experiment_controllers table.
-- id - experiment id (one condition variable is allocated per experiment)
-- result - condition variable name, or NULL if no such experiment
CREATE OR REPLACE FUNCTION experiment_controller_init_cond(varchar(32))
    RETURNS text AS '
    SELECT 'experiment_' || oid || '_ctrl_init' FROM experiments
    WHERE id = $1;
' LANGUAGE 'sql';

-- Get condition variable name for 'controller started' notifications.
-- These notifications are sent when queued controllers start up,
-- i.e., upon controller registration.
-- id - experiment id (one condition variable is allocated per experiment)
-- result - condition variable name, or NULL if no such experiment
CREATE OR REPLACE FUNCTION experiment_controller_start_cond(varchar(32))
    RETURNS text AS '
    SELECT 'experiment_' || oid || '_ctrl_start' FROM experiments
    WHERE id = $1;
' LANGUAGE 'sql';

```

```

-- Get condition variable name for 'controller stopped' notifications.
-- These notifications are sent when controllers are deleted from
-- the experiment_controllers table.
-- id - experiment id (one condition variable is allocated per experiment)
-- result - condition variable name, or NULL if no such experiment
CREATE OR REPLACE FUNCTION experiment_controller_stop_cond(vvarchar(32))
    RETURNS text AS '
    SELECT 'experiment_' || oid || '_ctrl_stop' FROM experiments
    WHERE id = $1;
' LANGUAGE 'sql';

-- Send experiment controller notifications. Experiment controller table
-- should never be batch-loaded, so don't assume any specific batch
-- loading event semantics.
CREATE OR REPLACE FUNCTION experiment_controllers_trigger() RETURNS OPAQUE AS '
    if {[info exists NEW(exp_id)]} {
        set qid "'[quote $NEW(exp_id)]'"
    } elseif {[info exists OLD(exp_id)]} {
        set qid "'[quote $OLD(exp_id)]'"
    } else {
        # this is probably an error, but who am I to judge
        return OK
    }
    if {[spi_exec \
        "SELECT experiment_controller_init_cond($qid) AS init_cond,
            experiment_controller_stop_cond($qid) AS stop_cond"] != 1} {
        # likewise
        return OK
    }
    if {![info exists init_cond]} {
        # likewise
        return OK
    }
    switch -exact $TG_op {
        INSERT {
            spi_exec "UPDATE experiments SET num_controllers = num_controllers+1
                WHERE id=$qid"
            spi_exec "NOTIFY $init_cond"
        }
        DELETE {
            spi_exec "NOTIFY $stop_cond"
        }
    }
    return OK
' LANGUAGE 'pltcl';
CREATE TRIGGER experiment_controllers_trigger
    AFTER INSERT OR DELETE
    ON experiment_controllers
    FOR EACH ROW EXECUTE PROCEDURE experiment_controllers_trigger();

```

```

-- Register an experiment controller. This function must be called by
-- the controller before creating any points.
-- id - controller id
CREATE OR REPLACE FUNCTION register_experiment_controller(integer)
  RETURNS integer AS '
  if {[spi_exec "UPDATE experiment_controllers SET pid=backend_pid()
                WHERE id=$1"] != 1} {
    elog ERROR "register_experiment_controller: controller $1 \\\
              does not exist"
    return_null
  }
  # broadcast registration event
  spi_exec "SELECT experiment_controller_start_cond(exp_id) AS start_cond
           FROM experiment_controllers WHERE id=$1"
  spi_exec "NOTIFY $start_cond"
  # ignore return value
  return 1
' LANGUAGE 'pltcl';

-- Create a fresh experiment point. This function must be called inside
-- of a transaction because it (indirectly) sends 'fresh point' notifications.
-- Such notifications must be delivered only when all experiment inputs for
-- the new point are loaded into the database.
-- id - controller id
-- result - sequence number of the newly created point
CREATE OR REPLACE FUNCTION fresh_experiment_point(integer) RETURNS integer AS '
# this is crude, but very effective
spi_exec "LOCK TABLE experiment_point_states IN SHARE ROW EXCLUSIVE MODE"
spi_exec "LOCK TABLE experiment_controllers IN SHARE ROW EXCLUSIVE MODE"
# find experiment id
if {[spi_exec "SELECT exp_id
              FROM experiment_controllers
              WHERE id=$1"] != 1} {
  elog ERROR "fresh_experiment_point: controller $1 does not exist"
  return_null
}
# find first unallocated seq_no
spi_exec "SELECT (CASE WHEN MAX(seq_no) IS NULL OR MAX(seq_no) < 0 THEN 0
                  ELSE MAX(seq_no) + 1
                END) AS seq_no
         FROM experiment_point_states WHERE id=''[quote $exp_id]'"
# mark experiment point
spi_exec "INSERT INTO experiment_point_states (id, seq_no, state)
         VALUES (''[quote $exp_id]'' , $seq_no, 0)"
return $seq_no
' LANGUAGE 'pltcl';

-- Unregister an experiment controller. The controller must not

```

```

-- create any points after this function has been called.
-- id - controller id
CREATE OR REPLACE FUNCTION unregister_experiment_ctrl(integer)
  RETURNS integer AS '
  # NOTICE: if there is an associated executor, foreign key constraint
  # will do the same thing when the executor is deleted; please maintain
  # this invariant if you change the system
  if {[spi_exec "DELETE FROM experiment_controllers WHERE id=$1"] != 1} {
    elog ERROR "unregister_experiment_ctrl: controller $1 does not exist"
    return null
  }
  # ignore return value
  return 1
' LANGUAGE 'pltcl';

-- Clean stale controller records.
-- result - number of stale controller records cleaned
CREATE OR REPLACE FUNCTION clean_experiment_controllers() RETURNS integer AS '
  SELECT SUM(unregister_experiment_ctrl(id))::integer AS result
  FROM experiment_controllers WHERE NOT backend_exists(pid);
' LANGUAGE 'sql';

-- Create some periodic cleanup triggers.  These triggers keep stale records
-- from accumulating indefinitely.  In particular, we guarantee that
-- all stale executor (resp. controller) records are removed before each
-- new executor (resp. controller) is inserted.  This works better than
-- you might think at first: we don't need to make space for new records
-- unless there are new records to insert.  However, these rules are by
-- no means sufficient to guard against pid wrap-arounds.  Since neither
-- the executor nor the controller table is synchronized with the OS
-- process table, these tables can store pids of dead processes.  If such
-- pid is recycled by the OS and assigned to a different backend, the
-- client (or another executor or controller) can mistake that backend
-- for the old one and hang waiting for events from a non-existent peer.
--
-- Therefore, executors/controllers should do the following before
-- registration:
-- SELECT SUM(unregister_experiment_executor(id))
--   FROM experiment_executors WHERE pid=backend_pid() AND id!=<my_id>;
-- SELECT SUM(unregister_experiment_ctrl(id))
--   FROM experiment_controllers WHERE pid=backend_pid() AND id!=<my_id>;
-- This is not done during registration to allow for multiple executors
-- or controllers in the same backend.
CREATE OR REPLACE FUNCTION experiment_executors_clean() RETURNS OPAQUE AS '
  spi_exec "SELECT clean_experiment_executors()"
  return [array get NEW]
' LANGUAGE 'pltcl';
CREATE OR REPLACE FUNCTION experiment_controllers_clean() RETURNS OPAQUE AS '
  spi_exec "SELECT clean_experiment_controllers()"
  return [array get NEW]

```

```

' LANGUAGE 'pltcl';
CREATE TRIGGER experiment_executors_clean
    BEFORE INSERT OR UPDATE ON experiment_executors
    FOR EACH ROW EXECUTE PROCEDURE experiment_executors_clean();
CREATE TRIGGER experiment_controllers_clean
    BEFORE INSERT OR UPDATE ON experiment_controllers
    FOR EACH ROW EXECUTE PROCEDURE experiment_controllers_clean();

-- Read/only experiment summaries.  These summaries exclude 'most of'
-- the stale executor and controller records, but the query alone does
-- not remove stale records from the tables (for performance reasons;
-- besides, stale records are sort of debug information).  'Most of'
-- means that OS pid reuse can introduce phantom executors and/or
-- controllers, but this is highly unlikely.  Eventually, executors
-- and controllers will figure things out, but the information may
-- be inconsistent for some time.
-- id - experiment id
-- description - experiment description
-- date - experiment creation date
-- np - number of processors per executor, if applicable
-- num_fresh_points - number of freshly loaded points
-- num_executing_points - number of executing points
-- num_computed_points - number of successfully computed points
-- num_crashed_points - number of points that crashed the simulation
-- num_controllers - total number of controllers ever successfully started
-- num_executors - total number of executors ever successfully started
-- num_queued_executors - number of queued executors (waiting for resources)
-- num_idle_executors - number of idle executors (waiting for inputs)
-- num_running_executors - number of running executors (computing inputs)
-- num_running_controllers - number of running controllers (in any state)
-- Example:
-- SELECT * FROM experiment_summaries ORDER BY date DESC;
CREATE VIEW experiment_summaries AS
    SELECT
        e.id AS id,
        e.description AS description,
        e.date AS date,
        e.np AS np,
        (SELECT COUNT(*) FROM experiment_point_states s
         WHERE s.id=e.id AND state=0)
        AS num_fresh_points,
        (SELECT COUNT(*) FROM experiment_point_states s
         WHERE s.id=e.id AND state=1 AND EXISTS
         (SELECT id FROM experiment_executors x
          WHERE x.exp_id=e.id AND x.exp_seq_no=s.seq_no AND
               backend_exists(x.pid)))
        AS num_executing_points,
        (SELECT COUNT(*) FROM experiment_point_states s
         WHERE s.id=e.id AND state=2)
        AS num_computed_points,
        (SELECT COUNT(*) FROM experiment_point_states s

```

```
WHERE s.id=e.id AND (state=3 OR (state=1 AND NOT EXISTS
  (SELECT id FROM experiment_executors x
    WHERE x.exp_id=e.id AND x.exp_seq_no=s.seq_no AND
          backend_exists(x.pid))))
  AS num_crashed_points,
e.num_controllers AS num_controllers,
e.num_executors AS num_executors,
(SELECT COUNT(*) FROM experiment_executors x
  WHERE x.exp_id=e.id AND state=0 AND backend_exists(x.pid))
  AS num_queued_executors,
(SELECT COUNT(*) FROM experiment_executors x
  WHERE x.exp_id=e.id AND state=1 AND backend_exists(x.pid))
  AS num_idle_executors,
(SELECT COUNT(*) FROM experiment_executors x
  WHERE x.exp_id=e.id AND state=2 AND backend_exists(x.pid))
  AS num_running_executors,
(SELECT COUNT(*) FROM experiment_controllers c
  WHERE c.exp_id=e.id AND
        (c.pid IS NULL OR backend_exists(c.pid)))
  AS num_running_controllers
FROM experiments e;
```

Appendix B

S⁴W Schemas

This Appendix contains the schemas for all S⁴W components.

B.1 S⁴WRT Schemas

```
-- Trigger for large object deletion.
-- col - column name of large object
CREATE OR REPLACE FUNCTION large_object_clean() RETURNS OPAQUE AS '
    if {[info exists OLD($1)]} {
        spi_exec "SELECT lo_unlink($OLD($1))"
    }
    return OK
' LANGUAGE 'pltcl';

-- Environments in Autocad's DXF format. This format is used for
-- bootstrapping only. Environments must be converted to S4W's native
-- XML format before they can be used with other S4W components.
-- See xml_environments for a list of environments in XML format.
-- id, seq_no - environment identification
-- dxf_lloid - oid of zlib-compressed DXF file
CREATE TABLE dxf_environments (
    id          varchar(32)    NOT NULL CHECK (id > ''),
    seq_no      integer        NOT NULL DEFAULT '0',
    dxf_lloid   oid            NOT NULL,
    PRIMARY KEY (id, seq_no)
);
CREATE TRIGGER dxf_environments_clean BEFORE DELETE ON dxf_environments
    FOR EACH ROW EXECUTE PROCEDURE large_object_clean('dxf_lloid');

-- Parameters to DXF to XML converter.
-- id, seq_no - experiment point identification
-- dxf_id, dxf_seq_no - reference to DXF environment
-- move_center - true to move coordinate origin to the environment center
-- add_floor - true to add a floor to the environment
-- add_ceiling - true to add a ceiling to the environment
-- elevation - wall height, or NULL if the environment is 3D
```

```

-- arclen - arc segment length for arc approximation, or NULL to ignore arcs
-- scale - scale factor (unit conversion), or NULL if the units are meters
CREATE TABLE dxf2xml_inputs (
    id          varchar(32)      NOT NULL CHECK (id > ''),
    seq_no      integer          NOT NULL DEFAULT '0',
    dxf_id      varchar(32)      NOT NULL,
    dxf_seq_no  integer          NOT NULL DEFAULT '0',
    move_center boolean         NOT NULL DEFAULT 'false',
    add_floor   boolean         NOT NULL DEFAULT 'false',
    add_ceiling boolean         NOT NULL DEFAULT 'false',
    elevation   float(8),
    arclen      float(8)         DEFAULT '0.1',
    scale       float(8),
    CHECK (elevation IS NULL OR elevation > 0),
    CHECK (scale IS NULL OR scale > 0),
    CHECK (arclen IS NULL OR arclen > 0),
    PRIMARY KEY (id, seq_no),
    FOREIGN KEY (id, seq_no)
        REFERENCES experiment_point_states (id, seq_no)
        ON DELETE CASCADE,
    FOREIGN KEY (dxf_id, dxf_seq_no)
        REFERENCES dxf_environments (id, seq_no)
        ON DELETE CASCADE
);
CREATE TRIGGER auto_init_dxf2xml_state
    BEFORE INSERT
    ON dxf2xml_inputs
    FOR EACH ROW EXECUTE PROCEDURE auto_init_point_state('id', 'seq_no');

-- Environments in custom XML format.  This format is the best one
-- for displaying environments on the screen.
-- id, seq_no - environment identification
-- x_min, x_max, y_min, y_max, z_min, z_max - environment bounds, in m
-- num_polygons - total number of polygons in the environment
-- num_vertices - total number of vertices in the environment
-- xml_loid - oid of zlib-compressed XML file
CREATE TABLE xml_environments (
    id          varchar(32)      NOT NULL,
    seq_no      integer          NOT NULL DEFAULT '0',
    x_min       float(8)         NOT NULL,
    x_max       float(8)         NOT NULL,
    y_min       float(8)         NOT NULL,
    y_max       float(8)         NOT NULL,
    z_min       float(8)         NOT NULL,
    z_max       float(8)         NOT NULL,
    num_polygons integer         NOT NULL CHECK (num_polygons >= 0),
    num_vertices integer         NOT NULL CHECK (num_vertices >= 0),
    xml_loid    oid              NOT NULL,
    PRIMARY KEY (id, seq_no),
    FOREIGN KEY (id, seq_no)
        REFERENCES dxf2xml_inputs (id, seq_no)

```

```

        ON DELETE CASCADE
    );
CREATE TRIGGER xml_environments_clean BEFORE DELETE ON xml_environments
    FOR EACH ROW EXECUTE PROCEDURE large_object_clean('xml_loid');

-- Parameters to triangulation.
-- id, seq_no - experiment point identification
-- xml_id, xml_seq_no - reference to XML environment
CREATE TABLE triang_inputs (
    id          varchar(32)      NOT NULL CHECK (id > ''),
    seq_no      integer          NOT NULL DEFAULT '0',
    xml_id      varchar(32)      NOT NULL,
    xml_seq_no  integer          NOT NULL DEFAULT '0',
    PRIMARY KEY (id, seq_no),
    FOREIGN KEY (id, seq_no)
        REFERENCES experiment_point_states (id, seq_no)
        ON DELETE CASCADE,
    FOREIGN KEY (xml_id, xml_seq_no)
        REFERENCES xml_environments (id, seq_no)
        ON DELETE CASCADE
);
CREATE TRIGGER auto_init_triang_state
    BEFORE INSERT
    ON triang_inputs
    FOR EACH ROW EXECUTE PROCEDURE auto_init_point_state('id', 'seq_no');

-- Environments in custom triangulated XML format.
-- id, seq_no - environment identification
-- num_triangles - total number of triangles in the environment
-- triang_loid - oid of zlib-compressed triangulated XML file
CREATE TABLE triang_environments (
    id          varchar(32)      NOT NULL,
    seq_no      integer          NOT NULL DEFAULT '0',
    num_triangles integer        NOT NULL,
    triang_loid oid              NOT NULL,
    CHECK (num_triangles >= 0),
    PRIMARY KEY (id, seq_no),
    FOREIGN KEY (id, seq_no)
        REFERENCES triang_inputs (id, seq_no)
        ON DELETE CASCADE
);
CREATE TRIGGER triang_environments_clean BEFORE DELETE ON triang_environments
    FOR EACH ROW EXECUTE PROCEDURE large_object_clean('triang_loid');

-- Parameters to octree space partitioning.
-- id, seq_no - experiment point identification
-- triang_id, triang_seq_no - reference to triangulated environment
-- min_dx - minimum voxel X dimension
-- max_triang - maximum number of triangles per voxel

```

```

-- max_dup - maximum degree of local triangle duplication
-- cscale - insertion scale factor, or zero for no triangles in int'l nodes
CREATE TABLE boct_inputs (
    id          varchar(32)    NOT NULL CHECK (id > ''),
    seq_no      integer        NOT NULL DEFAULT '0',
    triang_id   varchar(32)    NOT NULL,
    triang_seq_no integer      NOT NULL DEFAULT '0',
    min_dx      float(8)       NOT NULL DEFAULT '10',
    max_triang  integer        NOT NULL DEFAULT '8',
    max_dup     integer        NOT NULL DEFAULT '4',
    cscale      float(8)       NOT NULL DEFAULT '0.01',
CHECK (min_dx > 0), CHECK (max_triang > 0),
CHECK (max_dup >= 0 AND max_dup <= 8), CHECK (cscale >= 0),
    PRIMARY KEY (id, seq_no),
    FOREIGN KEY (id, seq_no)
        REFERENCES experiment_point_states (id, seq_no)
        ON DELETE CASCADE,
    FOREIGN KEY (triang_id, triang_seq_no)
        REFERENCES triang_environments (id, seq_no)
        ON DELETE CASCADE
);
CREATE TRIGGER auto_init_boct_state
    BEFORE INSERT
    ON boct_inputs
    FOR EACH ROW EXECUTE PROCEDURE auto_init_point_state('id', 'seq_no');

-- Environments in custom octree XML format.  This table only contains
-- octree statistics, not actual octree files.  Octree files are usually
-- too large to be stored in the database (or on disk for that matter).
-- The ray tracer regenerates octrees from data in boct_inputs.
-- id, seq_no - environment identification
-- avg_voxel_depth - average voxel depth, weighted by # triangles in voxel
-- max_voxel_depth - maximum voxel depth
-- num_internal - number of internal octree nodes
-- num_leaves - number of leaf octree nodes
-- num_empty - number of empty octree nodes
-- num_triangles - total number of triangles in the octree
-- avg_triangles_voxel - average number of triangles per voxel
-- max_triangles_voxel - maximum number of triangles per voxel
CREATE TABLE octree_environments (
    id          varchar(32)    NOT NULL,
    seq_no      integer        NOT NULL DEFAULT '0',
    avg_voxel_depth float(8)   NOT NULL CHECK (avg_voxel_depth >= 0),
    max_voxel_depth integer     NOT NULL CHECK (max_voxel_depth >= 0),
    num_internal integer        NOT NULL CHECK (num_internal >= 0),
    num_leaves  integer        NOT NULL CHECK (num_leaves >= 0),
    num_empty   integer        NOT NULL CHECK (num_empty >= 0),
    num_triangles integer       NOT NULL CHECK (num_triangles >= 0),
    avg_triangles_voxel float(8) NOT NULL CHECK (max_triangles_voxel >= 0),
    max_triangles_voxel integer NOT NULL CHECK (avg_triangles_voxel >= 0),
    PRIMARY KEY (id, seq_no),

```

```

        FOREIGN KEY (id, seq_no)
            REFERENCES boct_inputs (id, seq_no)
            ON DELETE CASCADE
    );

-- Antenna definitions.
-- id, seq_no - antenna identification
-- gain - maximum gain in direction (phi=0, theta=0), in dBi
-- type - 0 (isotropic), 1 (omnidirectional), 2 (biconical),
--       3 (waveguide), or 4 (pyramidal horn)
CREATE TABLE antennas (
    id          varchar(32)      NOT NULL CHECK (id > ''),
    seq_no      integer         NOT NULL DEFAULT '0',
    gain        float(8)        NOT NULL DEFAULT '0',
    type        integer         NOT NULL DEFAULT '0',
    CHECK (type >= 0 AND type <= 4),
    PRIMARY KEY (id, seq_no)
);

-- Commonly used generic antenna types.
INSERT INTO antennas (id, seq_no, gain, type)
    VALUES ('isotropic', '0', '0', '0');
INSERT INTO antennas (id, seq_no, gain, type)
    VALUES ('omnidirectional', '0', '0', '1');

-- Extra parameters for waveguide antennas
-- id, seq_no - antenna identification
-- width, height - antenna dimensions
CREATE TABLE waveguide_antennas (
    id          varchar(32)      NOT NULL,
    seq_no      integer         NOT NULL DEFAULT '0',
    width       float(8)        NOT NULL CHECK (width > 0),
    height      float(8)        NOT NULL CHECK (height > 0),
    FOREIGN KEY (id, seq_no) REFERENCES antennas (id, seq_no)
            ON DELETE CASCADE
);

-- Extra parameters for pyramidal horn antennas
-- id, seq_no - antenna identification
-- width, height - antenna dimensions
-- rw, rh - antenna cone radii in the two dimensions
CREATE TABLE horn_antennas (
    id          varchar(32)      NOT NULL,
    seq_no      integer         NOT NULL DEFAULT '0',
    width       float(8)        NOT NULL CHECK (width > 0),
    height      float(8)        NOT NULL CHECK (height > 0),
    rw          float(8)        NOT NULL CHECK (rw > 0),
    rh          float(8)        NOT NULL CHECK (rh > 0),
    FOREIGN KEY (id, seq_no) REFERENCES antennas (id, seq_no)
            ON DELETE CASCADE
);

```

```

-- Ray tracing parameters.
-- id, seq_no - ray tracing run identification
-- octree_id, octree_seq_no - reference to octree XML environment
-- max_transmissions - maximum number of transmissions (wall penetrations)
-- transmission_loss - transmission (wall penetration) loss, in dB
-- max_reflections - maximum number of reflections
-- reflection_loss - reflection loss, in dB
-- min_power - absolute power threshold, in dBW
-- power_range - relative threshold, in dB
-- path_loss_factor - free space path loss factor
CREATE TABLE s4wrt_inputs (
    id          varchar(32)  NOT NULL CHECK (id > ''),
    seq_no      integer      NOT NULL DEFAULT '0',
    octree_id   varchar(32)  NOT NULL,
    octree_seq_no integer    NOT NULL DEFAULT '0',
    max_transmissions integer NOT NULL DEFAULT '3',
    transmission_loss float(8) NOT NULL DEFAULT '12',
    max_reflections integer  NOT NULL DEFAULT '5',
    reflection_loss float(8) NOT NULL DEFAULT '6',
    min_power   float(8)     NOT NULL DEFAULT '-278',
    power_range float(8)     NOT NULL DEFAULT '35',
    path_loss_factor float(8) NOT NULL DEFAULT '1',
    CHECK (max_transmissions >= 0), CHECK (transmission_loss >= 0),
    CHECK (max_reflections >= 0), CHECK (reflection_loss >= 0),
    CHECK (power_range >= 0), CHECK (path_loss_factor >= 1),
    PRIMARY KEY (id, seq_no),
    FOREIGN KEY (id, seq_no)
        REFERENCES experiment_point_states (id, seq_no)
        ON DELETE CASCADE,
    FOREIGN KEY (octree_id, octree_seq_no)
        REFERENCES octree_environments (id, seq_no)
        ON DELETE CASCADE
);
CREATE TRIGGER auto_init_s4wrt_state
    BEFORE INSERT
    ON s4wrt_inputs
    FOR EACH ROW EXECUTE PROCEDURE auto_init_point_state('id', 'seq_no');

-- Transmitter locations.
-- s4wrt_id, s4wrt_seq_no - ray tracing run identification
-- id - transmitter location identification within the ray tracing run
-- seq_no - transmitter location order within the ray tracing run
-- x, y, z - Cartesian coordinates of the reference point, in m
-- freq - reference carrier frequency, in MHz
-- power - total transmit power, in dBW
-- tessl - geodesic tessellation frequency for each antenna
-- refdist - reference distance for path loss calculations, in m
-- antenna_id, antenna_seq_no - reference to antenna type
-- phi_a, theta_a - reference antenna orientation, in radians
CREATE TABLE s4wrt_transmitter_locations (

```

```

s4wrt_id      varchar(32)      NOT NULL,
s4wrt_seq_no  integer        NOT NULL DEFAULT '0',
id            varchar(32)    NOT NULL CHECK (id > ''),
seq_no       integer        NOT NULL DEFAULT '0',
x            float(8)       NOT NULL,
y            float(8)       NOT NULL,
z            float(8)       NOT NULL,
freq         float(8)       NOT NULL DEFAULT '900',
power        float(8)       NOT NULL DEFAULT '20',
tessl        integer        NOT NULL DEFAULT '100',
refdist      float(8)       NOT NULL DEFAULT '1',
antenna_id   varchar(32)    NOT NULL DEFAULT 'isotropic',
antenna_seq_no integer      NOT NULL DEFAULT '0',
phi_a        float(8)       NOT NULL DEFAULT '0',
theta_a      float(8)       NOT NULL DEFAULT '1.57',
CHECK (freq > 0), CHECK (tessl > 0), CHECK (refdist > 0),
PRIMARY KEY (s4wrt_id, s4wrt_seq_no, id),
FOREIGN KEY (s4wrt_id, s4wrt_seq_no)
    REFERENCES s4wrt_inputs (id, seq_no)
    ON DELETE CASCADE,
FOREIGN KEY (antenna_id, antenna_seq_no)
    REFERENCES antennas (id, seq_no)
    ON DELETE CASCADE
);

-- Transmitter location antennas.
-- s4wrt_id, s4wrt_seq_no - ray tracing run identification
-- txloc_id, ant_no - antenna identification within the ray tracing run
-- r - distance from reference location to antenna location, in m
-- phi, theta - direction from ref. location to antenna location, in radians
-- weight - relative power of transmit antenna (sum will be normalized by app.)
-- shift - shift from reference frequency to antenna frequency, in MHz
-- phi_a, theta_a - orientation shift from reference orientation, in radians
CREATE TABLE s4wrt_txloc_antennas (
    s4wrt_id      varchar(32)      NOT NULL,
    s4wrt_seq_no  integer          NOT NULL DEFAULT '0',
    txloc_id     varchar(32)      NOT NULL,
    ant_no       integer          NOT NULL DEFAULT '0',
    r            float(8)         NOT NULL DEFAULT '0',
    phi          float(8)         NOT NULL DEFAULT '0',
    theta        float(8)         NOT NULL DEFAULT '0',
    weight       float(8)         NOT NULL DEFAULT '1',
    shift        float(8)         NOT NULL DEFAULT '0',
    phi_a        float(8)         NOT NULL DEFAULT '0',
    theta_a      float(8)         NOT NULL DEFAULT '0',
    CHECK (r >= 0), CHECK (weight > 0),
    PRIMARY KEY (s4wrt_id, s4wrt_seq_no, txloc_id, ant_no),
    FOREIGN KEY (s4wrt_id, s4wrt_seq_no, txloc_id)
        REFERENCES s4wrt_transmitter_locations
            (s4wrt_id, s4wrt_seq_no, id)
    ON DELETE CASCADE
);

```

```

);

-- Receiver locations.
-- s4wrt_id, s4wrt_seq_no - ray tracing run identification
-- id - receiver location identification within the ray tracing run
-- seq_no - receiver location order within the ray tracing run
-- x, y, z - Cartesian coordinates of the reference point, in m
-- antenna_id, antenna_seq_no - reference to antenna type
-- phi_a, theta_a - reference antenna orientation, in radians
CREATE TABLE s4wrt_receiver_locations (
    s4wrt_id          varchar(32)      NOT NULL,
    s4wrt_seq_no     integer           NOT NULL DEFAULT '0',
    id               varchar(32)      NOT NULL CHECK (id > ''),
    seq_no          integer           NOT NULL DEFAULT '0',
    x               float(8)         NOT NULL,
    y               float(8)         NOT NULL,
    z               float(8)         NOT NULL,
    antenna_id      varchar(32)      NOT NULL DEFAULT 'isotropic',
    antenna_seq_no  integer           NOT NULL DEFAULT '0',
    phi_a           float(8)         NOT NULL DEFAULT '3.14',
    theta_a        float(8)         NOT NULL DEFAULT '1.57',
    PRIMARY KEY (s4wrt_id, s4wrt_seq_no, id),
    FOREIGN KEY (s4wrt_id, s4wrt_seq_no)
        REFERENCES s4wrt_inputs (id, seq_no)
        ON DELETE CASCADE,
    FOREIGN KEY (antenna_id, antenna_seq_no)
        REFERENCES antennas (id, seq_no)
        ON DELETE CASCADE
);

-- Receiver location antennas.
-- s4wrt_id, s4wrt_seq_no - ray tracing run identification
-- rxloc_id, ant_no - antenna identification within the ray tracing run
-- r - distance from reference location to antenna location, in m
-- phi, theta - direction from ref. location to antenna location, in radians
-- phi_a, theta_a - orientation shift from reference orientation, in radians
CREATE TABLE s4wrt_rxloc_antennas (
    s4wrt_id          varchar(32)      NOT NULL,
    s4wrt_seq_no     integer           NOT NULL DEFAULT '0',
    rxloc_id         varchar(32)      NOT NULL,
    ant_no           integer           NOT NULL DEFAULT '0',
    r               float(8)         NOT NULL DEFAULT '0',
    phi             float(8)         NOT NULL DEFAULT '0',
    theta           float(8)         NOT NULL DEFAULT '0',
    phi_a           float(8)         DEFAULT '0',
    theta_a        float(8)         DEFAULT '0',
    CHECK (r >= 0),
    PRIMARY KEY (s4wrt_id, s4wrt_seq_no, rxloc_id, ant_no),
    FOREIGN KEY (s4wrt_id, s4wrt_seq_no, rxloc_id)
        REFERENCES s4wrt_receiver_locations

```

```

                (s4wrt_id, s4wrt_seq_no, id)
            ON DELETE CASCADE
        );

-- Receiver location grids.
-- s4wrt_id, s4wrt_seq_no - ray tracing run identification
-- id - receiver location grid identification within the ray tracing run
-- seq_no - receiver location grid order within the ray tracing run
-- x_min, x_max, y_min, y_max - grid bounds, in m
-- z - grid elevation (z coordinate), in m
-- d - receiver location separation in X and Y dimensions, in m
-- antenna_id, antenna_seq_no - reference to antenna type
-- phi_a, theta_a - reference antenna orientation, in radians
CREATE TABLE s4wrt_receiver_location_grids (
    s4wrt_id          varchar(32)      NOT NULL,
    s4wrt_seq_no     integer           NOT NULL DEFAULT '0',
    id                varchar(32)      NOT NULL CHECK (id > ''),
    seq_no           integer           NOT NULL DEFAULT '0',
    x_min            float(8)          NOT NULL,
    x_max            float(8)          NOT NULL,
    y_min            float(8)          NOT NULL,
    y_max            float(8)          NOT NULL,
    z                float(8)          NOT NULL,
    d                float(8)          NOT NULL,
    antenna_id       varchar(32)      NOT NULL DEFAULT 'isotropic',
    antenna_seq_no   integer           NOT NULL DEFAULT '0',
    phi_a            float(8)          NOT NULL DEFAULT '3.14',
    theta_a          float(8)          NOT NULL DEFAULT '1.57',
    CHECK (x_min <= x_max AND y_min <= y_max),
    PRIMARY KEY (s4wrt_id, s4wrt_seq_no, id),
    FOREIGN KEY (s4wrt_id, s4wrt_seq_no)
        REFERENCES s4wrt_inputs (id, seq_no)
        ON DELETE CASCADE,
    FOREIGN KEY (antenna_id, antenna_seq_no)
        REFERENCES antennas (id, seq_no)
        ON DELETE CASCADE
);

-- Receiver location grid antennas.
-- s4wrt_id, s4wrt_seq_no - ray tracing run identification
-- rxloc_g_id, ant_no - antenna identification within the ray tracing run
-- r - distance from reference location to antenna location, in m
-- phi, theta - direction from ref. location to antenna location, in radians
-- phi_a, theta_a - orientation shift from reference orientation, in radians
CREATE TABLE s4wrt_rxloc_grid_antennas (
    s4wrt_id          varchar(32)      NOT NULL,
    s4wrt_seq_no     integer           NOT NULL DEFAULT '0',
    rxloc_g_id       varchar(32)      NOT NULL,
    ant_no           integer           NOT NULL DEFAULT '0',
    r                float(8)          NOT NULL DEFAULT '0',

```

```

    phi                float(8)          NOT NULL DEFAULT '0',
    theta              float(8)          NOT NULL DEFAULT '0',
    phi_a              float(8)          DEFAULT '0',
    theta_a            float(8)          DEFAULT '0',
    CHECK (r >= 0),
    PRIMARY KEY (s4wrt_id, s4wrt_seq_no, rxloc_g_id, ant_no),
    FOREIGN KEY (s4wrt_id, s4wrt_seq_no, rxloc_g_id)
        REFERENCES s4wrt_receiver_location_grids
            (s4wrt_id, s4wrt_seq_no, id)
    ON DELETE CASCADE
);

-- Receiver location PDPs (impulse responses).
-- s4wrt_id, s4wrt_seq_no - ray tracing run identification
-- txloc_id, txloc_ant_no - transmitter location antenna identification
-- rxloc_id, ant_no - receiver location antenna identification
-- pdp - raw power delay profile (impulse response)
CREATE TABLE s4wrt_rxloc_pdps (
    s4wrt_id            varchar(32)       NOT NULL,
    s4wrt_seq_no        integer           NOT NULL DEFAULT '0',
    txloc_id            varchar(32)       NOT NULL,
    txloc_ant_no        integer           NOT NULL DEFAULT '0',
    rxloc_id            varchar(32)       NOT NULL DEFAULT '0',
    rxloc_ant_no        integer           NOT NULL DEFAULT '0',
    pdp                 raw_pdp           NOT NULL,
    PRIMARY KEY (s4wrt_id, s4wrt_seq_no, txloc_id, txloc_ant_no,
                rxloc_id, rxloc_ant_no),
    FOREIGN KEY (s4wrt_id, s4wrt_seq_no, txloc_id, txloc_ant_no)
        REFERENCES s4wrt_txloc_antennas
            (s4wrt_id, s4wrt_seq_no, txloc_id, ant_no)
    ON DELETE CASCADE,
    FOREIGN KEY (s4wrt_id, s4wrt_seq_no, rxloc_id, rxloc_ant_no)
        REFERENCES s4wrt_rxloc_antennas
            (s4wrt_id, s4wrt_seq_no, rxloc_id, ant_no)
    ON DELETE CASCADE
);

-- Receiver location grid PDPs (impulse responses).
-- s4wrt_id, s4wrt_seq_no - ray tracing run identification
-- txloc_id, txloc_ant_no - transmitter location antenna identification
-- rxloc_g_id, ant_no, row, col - receiver location antenna identification
-- pdp - raw power delay profile (impulse response)
CREATE TABLE s4wrt_rxloc_grid_pdps (
    s4wrt_id            varchar(32)       NOT NULL,
    s4wrt_seq_no        integer           NOT NULL DEFAULT '0',
    txloc_id            varchar(32)       NOT NULL,
    txloc_ant_no        integer           NOT NULL DEFAULT '0',
    rxloc_g_id          varchar(32)       NOT NULL DEFAULT '0',
    rxloc_g_ant_no      integer           NOT NULL DEFAULT '0',
    row                 integer           NOT NULL CHECK (row >= 0),

```

```

col                integer                NOT NULL CHECK (col >= 0),
pdp                raw_pdp                NOT NULL,
PRIMARY KEY (s4wrt_id, s4wrt_seq_no, txloc_id, txloc_ant_no,
            rxloc_g_id, rxloc_g_ant_no, row, col),
FOREIGN KEY (s4wrt_id, s4wrt_seq_no, txloc_id, txloc_ant_no)
REFERENCES s4wrt_txloc_antennas
            (s4wrt_id, s4wrt_seq_no, txloc_id, ant_no)
ON DELETE CASCADE,
FOREIGN KEY (s4wrt_id, s4wrt_seq_no, rxloc_g_id, rxloc_g_ant_no)
REFERENCES s4wrt_rxloc_grid_antennas
            (s4wrt_id, s4wrt_seq_no, rxloc_g_id, ant_no)
ON DELETE CASCADE
);

```

B.2 WCDMA Simulation Schemas

```

-- WCDMA experiment inputs.
-- id, seq_no - experiment point identification
-- data_rate - downlink data rate, bps
-- no_fingers - number of fingers on rake receiver
-- no_inter - number of users on the same transmitter
-- fd - maximum Doppler frequency, Hz
-- pulse_shape - pulse shape
-- coding - type of forward error correction codes
-- hard_soft - decoding decision algorithm, usually soft
-- samples_chip - make this greater than one for more accurate results
-- noise_level - noise output of the receiver, in watts
-- no_frames - number of frames per BER estimate
-- max_frames - hard limit on # frames for statistically significant runs
-- confidence - how confident would you like to be in the results?
-- rel_error - how wide can the confidence interval be?
-- min_ber - what is the smallest ber value you care about?
-- channels - array of channels from different transmitter antennas
-- other - custom parameters, 'key1 value1 key2 value2 ...'
-- NOTE: no_inter has nothing to do with signals of interfering users.
-- The former is related to allocating orthogonal codes to different
-- users while the latter is related to propagation characteristics
-- of the environment.
-- NOTE: max_frames, rel_error, and min_ber only matter when confidence>0.
-- Batches of no_frames frames are simulated until either:
-- 1. The max_frames limit is reached or exceeded.
-- 2. We are confidence*100% confident that the true BER <= min_ber.
-- 3. The width of the confidence*100% confidence interval <= rel_error*BER.
-- (Note also that these rules say nothing about FER.)
CREATE TABLE wcdma_inputs (
    id                varchar(32)        NOT NULL,
    seq_no            integer             NOT NULL DEFAULT '0',
    data_rate         integer             NOT NULL DEFAULT '8000',
    no_fingers        integer             NOT NULL DEFAULT '3',
    no_inter          integer             NOT NULL DEFAULT '0',
    fd                float4              NOT NULL DEFAULT '5.0',

```

```

pulse_shape      varchar(4)      NOT NULL DEFAULT 'rect',
coding           varchar(4)      NOT NULL DEFAULT 'nc',
hard_soft       varchar(4)      NOT NULL DEFAULT 'soft',
samples_chip    integer         NOT NULL DEFAULT '5',
noise_level     float4          NOT NULL DEFAULT '1.0',
no_frames       integer         NOT NULL DEFAULT '10000',
max_frames      integer         NOT NULL DEFAULT '500000',
confidence      float4          NOT NULL DEFAULT '0.0',
rel_error       float4          NOT NULL DEFAULT '0.2',
min_ber        float4          NOT NULL DEFAULT '0.001',
channels        sampled_pdp[]   NOT NULL DEFAULT '{"(0,1,(0,1))"}',
other          text            NOT NULL DEFAULT '',
CHECK (data_rate >= 1),
CHECK (no_fingers >= 1 AND no_fingers <= 3),
CHECK (no_inter >= 0),
CHECK (fd >= 0),
CHECK (pulse_shape = 'rect' OR pulse_shape = 'rrc'),
CHECK (coding = 'nc' OR coding = '1/2' OR coding = '1/3'),
CHECK (hard_soft = 'hard' OR hard_soft = 'soft'),
CHECK (samples_chip >= 1),
CHECK (sttd_factor >= 0.0 AND sttd_factor <= 1.0),
CHECK (noise_level > 0.0),
CHECK (no_frames >= 1),
CHECK (max_frames >= 1),
CHECK (confidence >= 0 AND confidence < 1),
CHECK (rel_error > 0),
CHECK (min_ber >= 0 AND min_ber <= 1),
PRIMARY KEY (id, seq_no),
FOREIGN KEY (id, seq_no)
    REFERENCES experiment_point_states (id, seq_no)
    ON DELETE CASCADE
);
CREATE TRIGGER auto_init_wcdma_state
    BEFORE INSERT
    ON wcdma_inputs
    FOR EACH ROW EXECUTE PROCEDURE auto_init_point_state('id', 'seq_no');

-- WCDMA experiment outputs.  There will be more than one output
-- record per input record when statistically significant results
-- are requested.  See wcdma_outputs_summary if only one output
-- record per input record is desired.
-- Technically, output records that correspond to the same input
-- record are i.i.d. samples of mean BERs/FERs over no_frames
-- frames.  Such output records are Gaussian distributed as long
-- as no_frames is 'large enough'.
-- id, seq_no - experiment point identification
-- ber - estimate of bit error rate after no_frames frames, 0..1
-- fer - estimate of frame error rate after no_frames frames, 0..1
CREATE TABLE wcdma_outputs (
    id          varchar(32)      NOT NULL,
    seq_no     int              NOT NULL DEFAULT '0',

```

```

        ber            float(8)          NOT NULL,
        fer            float(8)          NOT NULL,
        CHECK (ber >= 0 AND ber <= 1),
        CHECK (fer >= 0 AND fer <= 1),
        FOREIGN KEY (id, seq_no)
            REFERENCES wcdma_inputs (id, seq_no)
            ON DELETE CASCADE
    );
CREATE INDEX wcdma_outputs_idx ON wcdma_outputs (id, seq_no);

-- WCDMA experiment outputs.  This view provides a convenient summary
-- of bit and frame error rate estimates for statistically significant
-- experiments.  (Actually, non-significant outputs are also included
-- to keep the interface uniform.  Include ber_var > 0 in the WHERE
-- clause to filter them out.  Also, no guarantees can be made about
-- the crashed points or the points being computed.  This is rarely
-- a problem, but you can join with experiment_point_states to filter
-- those out as well.)
-- id, seq_no - experiment point identification (sort of a primary key)
-- ber - estimate of bit error rate, 0..1
-- ber_var - sample variance of bit error rate, or zero if sample size is 1
-- fer - estimate of frame error rate, 0..1
-- fer_var - sample variance of frame error rate, or zero if sample size is 1
-- total_frames - total number of frames simulated to obtain these outputs
CREATE VIEW wcdma_outputs_summary AS
    SELECT
        i.id AS id,
        i.seq_no AS seq_no,
        AVG(o.ber) AS ber,
        VARIANCE(o.ber) AS ber_var,
        AVG(o.fer) AS fer,
        VARIANCE(o.fer) AS fer_var,
        SUM(i.no_frames) AS total_frames
    FROM wcdma_inputs i, wcdma_outputs o
    WHERE i.id = o.id AND i.seq_no = o.seq_no
    GROUP BY i.id, i.seq_no;

```

B.3 VTDIRECT Schemas

```

-- Note on user-defined conversion functions:
-- We use function names, not function oids, because names make it easier
-- to change function definitions in the database.  However, there is a
-- performance penalty for using names.  Also, keep in mind that a major
-- change in a function definition will invalidate the results computed
-- using the old definition.  Using function names in the tables does not
-- allow the database to detect that the function definition has changed.
-- It is up to the user to update whatever data is necessary after changing
-- function definition.  (That is, it is OK to fix bugs in function
-- definitions, but you probably don't want to go beyond that.)

```

```

-- Compute shortfall of x w.r.t. the threshold t.
-- x - value
-- t - threshold
-- result - t-x if t>x, or zero otherwise
CREATE OR REPLACE FUNCTION shortfall(float4, float4) RETURNS float4 AS '
    SELECT (CASE WHEN $1 < $2 THEN $2 - $1 ELSE 0.0 END)::float4;
' LANGUAGE 'sql' WITH (isstrict);

-- Make sure that a function can be used as an objective function.
-- This subroutine aborts in case of an error or returns true in
-- case of success. An objective function must take two or more
-- arguments: experiment id (varchar or text), sequence number (int4),
-- and zero or more of other arguments that depend on the function.
-- An objective function must compute a single aggregate value over
-- all outputs of the experiment point (id, seq_no) and return this
-- value as float4. Small numbers should correspond to good outcomes,
-- large numbers should correspond to bad outcomes.
-- name - objective function name
CREATE OR REPLACE FUNCTION objective_fn_check(name) RETURNS boolean AS '
    ... lengthy source code omitted ...
' LANGUAGE 'pltcl';

-- Make sure that a function can be used as an objective function.
-- objective_fn - objective function id
-- experiment - experiment id
-- result - objective function name
CREATE OR REPLACE FUNCTION objective_fn_check(varchar(32), varchar(32))
    RETURNS name AS '
    ... lengthy source code omitted ...
' LANGUAGE 'pltcl';

-- Evaluate an objective function at an experiment point. Outputs
-- for this point (or a significant portion thereof) should have
-- been computed, or else the results may be unexciting.
-- id - objective function id
-- exp_id, exp_seq_no - experiment point to evaluate
-- data - Tcl list of extra arguments to the objective function
CREATE OR REPLACE FUNCTION objective_fn_eval(varchar(32), varchar(32), int4,
    text) RETURNS float4 AS '
    spi_exec "SELECT objective_fn_check('[quote $1]''', '[quote $2]''')
        AS objective_fn"
    if {[llength $4] <= 0} {
        spi_exec "SELECT ${objective_fn}('[quote $2]''', $3) AS r"
    } else {
        spi_exec "SELECT ${objective_fn}('[quote $2]''', $3,
            '[join [quote $4] ''', ''']') AS r"
    }
    if {[info exists r]} {

```

```

        return $r
    } else {
        return_null
    }
' LANGUAGE 'pltcl';

-- Make sure that a function can be used as a bind function.
-- This subroutine aborts in case of an error or returns true in
-- case of success. A bind function must take three or more
-- arguments: experiment id (varchar or text), sequence number (int4),
-- parameter value (float4), and zero or more of other arguments that
-- depend on the function. A bind function must bind the value (third
-- argument) to some parameter of the experiment point (id, seq_no).
-- (The point will be created before the bind function is called.)
-- The bind function must return true if the parameter was updated
-- or false if no parameter record existed. There will usually be
-- one bind function per controllable value of an experiment.
-- name - bind function name
CREATE OR REPLACE FUNCTION bind_fn_check(name) RETURNS boolean AS '
    ... lengthy source code omitted ...
' LANGUAGE 'pltcl';

-- Make sure that a function can be used as a bind function.
-- bind_fn - bind function id
-- experiment - experiment id
-- result - bind function name
CREATE OR REPLACE FUNCTION bind_fn_check(varchar(32), varchar(32))
    RETURNS name AS '
    ... lengthy source code omitted ...
' LANGUAGE 'pltcl';

-- Bind a value to an experiment parameter. This function is used to
-- update a value of a single parameter, not to create a new experiment
-- point. Therefore, the experiment point being updated must already
-- exist. This function aborts if the point does not exist, or returns
-- true otherwise.
-- id - objective function id
-- exp_id, exp_seq_no - experiment point to update
-- value - parameter value
-- data - Tcl list of extra arguments to the bind function
CREATE OR REPLACE FUNCTION bind_fn_eval(varchar(32), varchar(32), int4,
    float4, text) RETURNS boolean AS '
    spi_exec "SELECT bind_fn_check('[quote $1]''', '[quote $2]''')
        AS bind_fn"
    if {[llength $5] <= 0} {
        spi_exec "SELECT ${bind_fn}('[quote $2]''', $3, $4) AS r"
    } else {
        spi_exec "SELECT ${bind_fn}('[quote $2]''', $3, $4,
            '[join [quote $5] ''', ''']''') AS r"
    }

```

```

    }
    if {[info exists r]} {
        if {[string match t* $r]} {
            return true
        }
    }
    elog ERROR "bind_fn_eval: point ($2,$3) does not exist"
    return_null
' LANGUAGE 'pltcl';

-- Make sure that a function can be used as a clone function.
-- This subroutine aborts in case of an error or returns true in
-- case of success. A clone function must take four or more
-- arguments: source experiment id (varchar or text), source sequence
-- number (int4), destination experiment id (varchar or text),
-- destination sequence number (int4), and zero or more of other
-- arguments that depend on the function. A clone function must copy
-- all experiment inputs (but not outputs) from the source point to
-- the destination point.
-- name - clone function name
CREATE OR REPLACE FUNCTION clone_fn_check(name) RETURNS boolean AS '
    ... lengthy source code omitted ...
' LANGUAGE 'pltcl';

-- Make sure that a function can be used as a clone function.
-- objective_fn - clone function id
-- experiment - experiment id
-- result - clone function name
CREATE OR REPLACE FUNCTION clone_fn_check(varchar(32), varchar(32))
    RETURNS name AS '
    ... lengthy source code omitted ...
' LANGUAGE 'pltcl';

-- Clone an experiment point. The source point must already exist.
-- If the destination point exists, it will be silently deleted.
-- This function aborts if the source point does not exist, or
-- returns true otherwise.
-- id - clone function id
-- src_id, src_seq_no - experiment point to clone
-- dst_id, dst_seq_no - experiment point to overwrite or create
-- data - Tcl list of extra arguments to the clone function
CREATE OR REPLACE FUNCTION clone_fn_eval(varchar(32), varchar(32), int4,
    varchar(32), int4, text) RETURNS boolean AS '
    spi_exec "SELECT
        clone_fn_check('[quote $1]',' '[quote $2]') AS clone_fn,
        clone_fn_check('[quote $1]',' '[quote $4]')"
    if {[llength $6] <= 0} {
        spi_exec "SELECT ${clone_fn}('[quote $2]',' $3, '[quote $4]','
            $5) AS r"

```

```

    } else {
        spi_exec "SELECT ${clone_fn}('[quote $2]'' , $3, '[quote $4]'' ,
            $5, '[join [quote $6] '' , '' ]'' ) AS r"
    }
    if {[info exists r]} {
        if {[string match -nocase t* $r]} {
            return true
        }
    }
    elog ERROR "clone_fn_eval: point ($2,$3) does not exist"
    return_null
' LANGUAGE 'pltcl';

-- Objective functions (used primarily for optimization).
-- id, description, date - objective function identification
-- component - component to which the function applies
-- objective_fn - function name
CREATE TABLE objective_functions (
    id            varchar(32)    NOT NULL CHECK (id > ''),
    description   text          NOT NULL DEFAULT '',
    date         date          NOT NULL DEFAULT CURRENT_DATE,
    component     varchar(32)    NOT NULL,
    objective_fn  name          NOT NULL,
    PRIMARY KEY (id),
    FOREIGN KEY (component)
        REFERENCES components (id)
        ON DELETE CASCADE
);
CREATE OR REPLACE FUNCTION objective_functions_check() RETURNS OPAQUE AS '
    if {[info exists NEW(objective_fn)]} {
        spi_exec "SELECT objective_fn_check('[quote $NEW(objective_fn)]'' )"
    }
    return [array get NEW]
' LANGUAGE 'pltcl';
CREATE TRIGGER objective_functions_check
    BEFORE INSERT OR UPDATE ON objective_functions
    FOR EACH ROW EXECUTE PROCEDURE objective_functions_check();

-- Bind functions (can be used for any iteration).
-- id, description, date - bind function identification
-- component - component to which the function applies
-- bind_fn - function name
CREATE TABLE bind_functions (
    id            varchar(32)    NOT NULL CHECK (id > ''),
    description   text          NOT NULL DEFAULT '',
    date         date          NOT NULL DEFAULT CURRENT_DATE,
    component     varchar(32)    NOT NULL,
    bind_fn      name          NOT NULL,
    PRIMARY KEY (id),
    FOREIGN KEY (component)

```

```

REFERENCES components (id)
ON DELETE CASCADE
);
CREATE OR REPLACE FUNCTION bind_functions_check() RETURNS OPAQUE AS '
  if {[info exists NEW(bind_fn)]} {
    spi_exec "SELECT bind_fn_check('[quote $NEW(bind_fn)]'')"
  }
  return [array get NEW]
' LANGUAGE 'pltcl';
CREATE TRIGGER bind_functions_check
BEFORE INSERT OR UPDATE ON bind_functions
FOR EACH ROW EXECUTE PROCEDURE bind_functions_check();

-- Clone functions (can be used for any iteration).
-- id, description, date - clone function identification
-- component - component to which the function applies
-- clone_fn - function name
CREATE TABLE clone_functions (
  id          varchar(32)      NOT NULL CHECK (id > ''),
  description text            NOT NULL DEFAULT '',
  date        date            NOT NULL DEFAULT CURRENT_DATE,
  component   varchar(32)      NOT NULL,
  clone_fn    name            NOT NULL,
  PRIMARY KEY (id),
  FOREIGN KEY (component)
    REFERENCES components (id)
    ON DELETE CASCADE
);
CREATE OR REPLACE FUNCTION clone_functions_check() RETURNS OPAQUE AS '
  if {[info exists NEW(clone_fn)]} {
    spi_exec "SELECT clone_fn_check('[quote $NEW(clone_fn)]'')"
  }
  return [array get NEW]
' LANGUAGE 'pltcl';
CREATE TRIGGER clone_functions_check
BEFORE INSERT OR UPDATE ON clone_functions
FOR EACH ROW EXECUTE PROCEDURE clone_functions_check();

-- Optimization parameters.
-- id, seq_no - optimization run identification
-- experiment - experiment for objective function evaluation
-- objective_fn - primary objective function
-- objective_fn_p - extra parameters to primary objective function
-- objective_fn2 - supplementary objective function (optional)
-- objective_fn2_p - extra parameters to supplementary objective function
-- clone_fn - point copy function
-- clone_fn_p - extra parameters to point copy function
-- max_iter - maximum number of iterations of DIRECT
-- max_eval - maximum number of objective function evaluations
-- min_diam - minimum box diameter

```

```

-- obj_conv - relative convergence threshold
CREATE TABLE opt_parameters (
    id          varchar(32)    NOT NULL,
    seq_no      integer        NOT NULL,
    experiment  varchar(32)    NOT NULL,
    objective_fn varchar(32)    NOT NULL,
    objective_fn_p text        NOT NULL DEFAULT '',
    objective_fn2 varchar(32),
    objective_fn2_p text        NOT NULL DEFAULT '',
    clone_fn    varchar(32)    NOT NULL,
    clone_fn_p  text          NOT NULL DEFAULT '',
    max_iter    integer        NOT NULL DEFAULT '10',
    max_eval    integer        NOT NULL DEFAULT '50',
    min_diam    float4         NOT NULL DEFAULT '0',
    obj_conv    float4         NOT NULL DEFAULT '0.0001',
    PRIMARY KEY (id, seq_no),
    FOREIGN KEY (id, seq_no)
        REFERENCES experiment_point_states (id, seq_no)
        ON DELETE CASCADE,
    FOREIGN KEY (experiment)
        REFERENCES experiments (id)
        ON DELETE CASCADE,
    FOREIGN KEY (objective_fn)
        REFERENCES objective_functions (id)
        ON DELETE CASCADE,
    FOREIGN KEY (objective_fn2)
        REFERENCES objective_functions (id)
        ON DELETE SET NULL
);

CREATE OR REPLACE FUNCTION opt_parameters_check() RETURNS OPAQUE AS '
    if {[info exists NEW(experiment)]} {
        if {[info exists NEW(objective_fn)]} {
            spi_exec "SELECT objective_fn_check(
                ''[quote $NEW(objective_fn)]'', ''[quote $NEW(experiment)]'')"
        }
        if {[info exists NEW(objective_fn2)]} {
            spi_exec "SELECT objective_fn_check(
                ''[quote $NEW(objective_fn2)]'', ''[quote $NEW(experiment)]'')"
        }
        if {[info exists NEW(clone_fn)]} {
            spi_exec "SELECT clone_fn_check(
                ''[quote $NEW(clone_fn)]'', ''[quote $NEW(experiment)]'')"
        }
    }
    return [array get NEW]
' LANGUAGE 'pltcl';

CREATE TRIGGER opt_parameters_check
    BEFORE INSERT OR UPDATE ON opt_parameters
    FOR EACH ROW EXECUTE PROCEDURE opt_parameters_check();

CREATE TRIGGER auto_init_opt_state
    BEFORE INSERT
    ON opt_parameters

```

```

FOR EACH ROW EXECUTE PROCEDURE auto_init_point_state('id', 'seq_no');

-- Optimization variables.  The values of these variables are controlled
-- by the optimizer.
-- id, seq_no - optimization run identification
-- variable - variable identification within the run
-- bind_fn - binds variable value to experiment point parameter(s)
-- bind_fn_p - extra parameters to bind function
-- min_val, max_val - hard bounds on variable values (inclusive)
CREATE TABLE opt_variables (
    id          varchar(32)    NOT NULL,
    seq_no      integer        NOT NULL,
    variable    varchar(32)    NOT NULL,
    bind_fn     varchar(32)    NOT NULL,
    bind_fn_p   text           NOT NULL DEFAULT '',
    min_val     float4         NOT NULL,
    max_val     float4         NOT NULL,
    CHECK (min_val < max_val),
    PRIMARY KEY (id, seq_no, variable),
    FOREIGN KEY (bind_fn)
        REFERENCES bind_functions (id)
        ON DELETE CASCADE,
    FOREIGN KEY (id, seq_no)
        REFERENCES opt_parameters (id, seq_no)
        ON DELETE CASCADE
);
CREATE OR REPLACE FUNCTION opt_variables_check() RETURNS OPAQUE AS '
    if {[info exists NEW(id)] && [info exists NEW(seq_no)] &&
        [spi_exec "SELECT experiment FROM opt_parameters
            WHERE id=''[quote $NEW(id)]'' AND
                seq_no=$NEW(seq_no)"] == 1} {
        if {[info exists NEW(bind_fn)]} {
            spi_exec "SELECT bind_fn_check(
                ''[quote $NEW(bind_fn)]'', ''[quote $experiment]'')"
        }
    }
    return [array get NEW]
' LANGUAGE 'pltcl';
CREATE TRIGGER opt_variables_check
    BEFORE INSERT OR UPDATE ON opt_variables
    FOR EACH ROW EXECUTE PROCEDURE opt_variables_check();

-- A log of optimization variable values.
-- Strictly speaking, iter_no should reference controlled experiment's
-- seq_no, but we don't enforce it.  If controlled experiment's outputs
-- are large, users may want to clean them up but keep the optimization
-- log for a bit longer.
-- id, seq_no, variable - variable identification
-- iter_no - iteration number
-- value - variable value

```

```
CREATE TABLE opt_variable_values (  
    id          varchar(32)    NOT NULL,  
    seq_no      integer        NOT NULL,  
    variable    varchar(32)    NOT NULL,  
    iter_no     integer        NOT NULL,  
    value       float4         NOT NULL,  
    PRIMARY KEY (id, seq_no, variable, iter_no),  
    FOREIGN KEY (id, seq_no, variable)  
        REFERENCES opt_variables (id, seq_no, variable)  
        ON DELETE CASCADE  
);  
  
-- A log of objective function values.  
-- id, seq_no - optimization run identification  
-- iter_no - iteration number  
-- value - value of primary objective function  
-- value2 - value of supplementary objective function (optional)  
CREATE TABLE opt_objective_values (  
    id          varchar(32)    NOT NULL,  
    seq_no      integer        NOT NULL,  
    iter_no     integer        NOT NULL,  
    value       float4         NOT NULL,  
    value2      float4,  
    PRIMARY KEY (id, seq_no, iter_no),  
    FOREIGN KEY (id, seq_no)  
        REFERENCES opt_parameters (id, seq_no)  
        ON DELETE CASCADE  
);
```

Appendix C

BSML DTD

```
<!ENTITY % boolean "(true|false|t|f|yes|no|y|n)">

<!-- attributes of primitive types:
  min - minimum value or string length (inclusive)
  max - maximum value or string length (inclusive)
  number - true means NaN is not allowed (doubles only)
  finite - true means +/-infinity is not allowed (doubles only)
  units - units for this type (doubles only)
-->
<!ENTITY % type_attributes "
  min          CDATA          #IMPLIED
  max          CDATA          #IMPLIED
  number       %boolean;     #IMPLIED
  finite       %boolean;     #IMPLIED
  units       CDATA          #IMPLIED
">

<!-- what schemas and schema blocks are composed of -->
<!ENTITY % schema_contents "
  (element | sequence | selection | repetition)
">
<!ENTITY % block_contents "
  (%schema_contents; | default | ref | code)
">

<!-- a collection of schemas -->
<!ELEMENT schemas ((description)?, (type | schema)*)>
<!ATTLIST schemas>

<!-- primitive type: attributes above and an optional
enumeration of legal values; derivation works by restriction;
builtin base types are: integer, string, double, boolean -->
<!ELEMENT type ((description)?, (values)?)>
<!ATTLIST type
  id          CDATA          #REQUIRED
  base       CDATA          #REQUIRED
  %type_attributes;
```

```

>
<!-- enumeration of legal values, no value is legal if empty -->
<!ELEMENT values ((value)*)>
<!ATTLIST values>
<!ELEMENT value (#PCDATA)>
<!ATTLIST value>

<!-- schema -->
<!ELEMENT schema ((description)?, (code)*, (%schema_contents;), (code)*)>
<!ATTLIST schema
    id          CDATA          #REQUIRED
>

<!-- an element can contain either
(a) character data of a primitive type (type attribute is present),
(b) zero or more schema blocks (type attribute is absent), or
(c) when type='*', any contents.
-->
<!ELEMENT element ((description)?, (attribute)*,
    ((values)? | (%block_contents;)*))>
<!ATTLIST element
    name        CDATA          #REQUIRED
    id          CDATA          #IMPLIED
    optional    %boolean;      "false"
    type        CDATA          #IMPLIED
    %type_attributes;
    default     CDATA          #IMPLIED
>

<!-- an attribute must contain a value of some primitive type -->
<!ELEMENT attribute ((description)?, (values)?)>
<!ATTLIST attribute
    name        CDATA          #REQUIRED
    id          CDATA          #IMPLIED
    type        CDATA          "string"
    %type_attributes;
    default     CDATA          #IMPLIED
>

<!-- a sequence is just a grouping, for convenience -->
<!ELEMENT sequence ((description)?, (%block_contents;)*>
<!ATTLIST sequence
    id          CDATA          #IMPLIED
    optional    %boolean;      "false"
>

<!-- a selection denotes a mutually exclusive choice of contents -->
<!ELEMENT selection ((description)?, (%block_contents;)+>
<!ATTLIST selection
    id          CDATA          #IMPLIED
    optional    %boolean;      "false"

```

```
>
<!-- a repetition denotes [min..max] repetitions of contents -->
<!ELEMENT repetition ((description)?, (%block_contents)*)>
<!ATTLIST repetition
  id          CDATA          #IMPLIED
  optional    %boolean;      "false"
  min         CDATA          "0"
  max         CDATA          "inf"
>

<!-- a reference to some block id in this schema,
or to an id of a different schema -->
<!ELEMENT ref ((description)?)>
<!ATTLIST ref
  id          CDATA          #REQUIRED
>

<!-- user code; language and component attributes facilitate
schema reuse (different components can have the same schema,
but different binding codes) -->
<!ELEMENT code (#PCDATA)>
<!ATTLIST code
  language    CDATA          #IMPLIED
  component   CDATA          #IMPLIED
>

<!-- default contents must conform to BSML schema block -->
<!ELEMENT default ANY>

<!-- XHTML usually goes here -->
<!ELEMENT description ANY>
```

Vita

Alex A. Verstak was born on November 12, 1979 in Minsk, Belarus. In 1996–1997, he attended the Belarusian State University majoring in Mechanics and Mathematics. Upon his move to Richmond, VA in 1997, Alex enrolled in J. Sargeant Reynolds Community College and thereafter transferred to Virginia Tech in 1998. He graduated *Magna Cum Laude* with a Bachelor of Science degree in Computer Science in May 2000.

Alex enrolled in the Master of Science program in Computer Science and Applications at Virginia Tech in Fall 2000. His primary research topics were data modeling and data mining for a problem solving environment targeted at wireless system researchers. He also did work in wireless propagation modeling using ray tracing on a Beowulf cluster of workstations. During his graduate studies, Alex did a summer internship with Trilogy, Inc. where he developed performance testing software for a data-intensive application in auto insurance.

Alex has been actively involved with the Association for Computing Machinery Student Chapter at Virginia Tech. He has participated in the annual ACM Mid-Atlantic Regional Programming Contest in 1998–2001 and the ACM International Collegiate Programming Contest in 2002. Alex was the President of the ACM Student Chapter at Virginia Tech in 2000–2001 and the Treasurer in 2001–2002.