Dynamic Workflows and Advanced Data Management for Problem Solving Environments

Dan Moisa
Department of Computer Science
Virginia Tech, Blacksburg, VA 24061

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

Examining Committee:

Naren Ramakrishnan, Chair
Calvin J. Ribbens
Srinidhi Varadarajan

May 7, 2004
Blacksburg, Virginia

Keywords: Dynamic Workflows, Data Harvesting, Problem Solving Environments, Data Transformations

Dynamic Workflows and Advanced Data Management for Problem Solving Environments

Dan Moisa

Abstract

Workflow management in problem solving environments (PSEs) is an emerging topic that aims to combine both data-oriented and execution-oriented views of scientific experiments, and closely integrate the processes underlying the practice of computational science with the software artifacts constituted by the PSE. This thesis presents a workflow management solution called **BREW** (BetteR Experiments through Workflow management) that provides functionality along four dimensions: components and installation management, experiment execution management, data management, and (full fledged) workflow management. BREW builds upon EMDAG, a first generation experiment management system designed at Virginia Tech which provided rudimentary facilities for supporting (only) the first two functionalities. BREW provides a complete dynamic workflow management solution wherein the PSE user can compose arbitrary scientific experiments and specify intended dynamic behavior of these experiments to an extent not previously possible. Along with the design details of the BREW system, this thesis identifies important tradeoffs underlying workflow management for PSEs, and presents two case studies involving large-scale data assimilation in bioinformatics experiments.

Acknowledgements

First I'd like to thank Naren Ramakrishnan, my advisor, for his guidance and support in getting me through all the stages of my thesis. His help is what made my thesis possible.

Alex Verstak is the creator of EMDAG, the system that BREW is based on, and his assistance was vital in implementing BREW. Greg Grothaus wrote the library for web batch browsing which BREW uses for experiments. And finally I'd like to thank all the other grad students for their supportive work and advice.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Problem solving environments (PSEs) have been a focused area of research in computational science for the last 10 years. While PSEs in the early 1990s were custom built for targeted domains by individual groups of researchers, today's systems are becoming more and more componentized and use modularized software architectures for realizing their functionality. This emphasis on componentization is increasingly buttressed by the rapid growth of grids, clusters, and other computing environments that encourage a distributed and loosely coupled methodology of software design.

An area that has become popular in PSE research is *workflow management*, owing to its combinational approach towards the classic data-oriented and execution-oriented views of scientific experiments. A workflow management system helps closely integrate the processes underlying the practice of computational science with the software artifacts constituted by the PSE. One example of such a system is ZENTURIO [15] which aims to model computation over parallel clusters and grid platforms. It allows the experimenter to define computational resources, identify components that perform the required data transformations, and specify a composition for achieving the flow of execution. While these traits are central to almost all workflow management systems, they differ in how they provide their functionality and the design tradeoffs they make.

1.1 In this Thesis

The main contribution of this thesis is the design and implementation of a workflow management system for scientific PSEs called **BREW** (BetteR Experiments through Workflow management). As part of BREW's specification, the thesis also defines a set of mandatory features that any workflow management system in a scientific PSE context must implement. These functionalities are organized into four categories: components and installation management, experiment execution management, data management, and (full fledged) workflow management. BREW builds upon EMDAG, a first generation experiment management system designed at Virginia Tech [20] which provided rudimentary facilities for supporting (only) the first two functionalities. BREW provides a complete dynamic workflow management solution wherein the PSE user can compose arbitrary scientific experiments and specify intended dynamic behavior of these experiments to an extent not previously possible.

BREW is designed to handle a large variety of experimental contexts in computational science. Applications composed from many individual components, each with varying input and output specifications, organized along complex execution graphs, benefit the most from management via BREW. Experiments that perform information retrieval from heterogenous Internet data sources are also a domain targeted by BREW, together with subsequent data transformation and reasoning experiments.

1.2 Reader's Guide

The thesis begins by introducing definitions and concepts vital to the understanding of the following chapters. With the aid of these definitions, Chapter 2 presents a set of requirements necessary for any modern workflow management system. Together with this set of features it also briefly presents eight previously developed systems in the area of workflow and experiment management. This presentation is focused on identifying the key design elements and functionality of each individual system, in order to clearly define what is lacking in the field of dynamic workflow management for problem solving environments.

Chapter 3 focuses on presenting the low-level design decisions and basic functionality of BREW. After introducing common schema notations that are used throughout the thesis, Chapter 3 proceeds with presenting the system's basic management structures for simple experiments. The connections with the EMDAG experiment management system are established, together with the modifications and additions introduced by BREW. A detailed walkthrough through simple experiment designs and their schemas is provided to facilitate the understanding of BREW's basic functionality. The chapter concludes by presenting a list of special cases and an overview of the basic features.

Chapter 4 expands on the basic functionality presented in Chapter 3. Complex experiment are now the focus of the discussion, together with any failure cases that might appear. This chapter also details dynamic workflow management functionality, and includes example experiments as well. After establishing the advanced capabilities of BREW, the chapter details several design constraints originating from the underlying software implementation choices.

Although BREW is targeted for many computational science contexts, two case studies in the field of bioinformatics are pursued in Chapter 5. Both examples contain full description of the complex experiments and the constituent simple experiments. The problem studied in each of the cases is thoroughly described, together with explanations about the various design decisions taken in building specific parts. Both case studies are concluded with specific results extracted from running the experiments designed with BREW; performance measurements and comparison to existing solutions further emphasize the benefits of using BREW.

The thesis concludes with a comparison of the workflow management systems presented throughout the discussion. This comparison is focused on clearly identifying the capabilities of each system in terms of the feature set introduced in Chapter 2, as well as identifying future work that can be performed on BREW.

For ease of understanding, the appendices also provides working BREW code for all the experiments and most of the features presented in the main chapters. The code is organized in three appendixes, the first presenting the sample experiments' code, the second focusing on BREW system functions and schemas, and the third detailing a third-party system (called NetScrape) used to achieve batch web browsing functionality (useful in the bioinformatics case studies).

Chapter 2

Background

This chapter first defines terminology used in the remainder of the thesis. We borrow some concepts and ideas from the literature on grid computing environments [16] although the motivating applications of BREW are different. A list of essential features for workflow management in PSEs are then presented, followed by a survey of related research. This chapter concludes with a discussion of common shortcomings and suggestions of what is required to realize a complete management system for scientific workflows.

2.1 Some Definitions

BREW's target application domain is broad, involving both executing programs on parallel clusters (e.g., compute-intensive simulations), as well as data-driven computational science (e.g., batch harvesting and assimilation of data and information across the web, in bioinformatics applications). To standardize the understanding of usage contexts, BREW utilizes the following terminology (examples are provided alongside each definition):

component : any piece of software or executable together with input and output descriptions, that helps the scientist to formalize the process of modeling a computation or data access.

component$_1$: **int factorial(int)**

experiment point : the smallest unit of data conforming to the input description of a component.

experiment_point$_1$: **3**

schema : a structure that conforms to either the input or output descriptions of a component. In the below example, if **a** denotes a container for integers, then *schema*$_1$ can be used as the input schema for *component*$_1$ above.

schema$_1$: **a**

simple experiment : a component associated with conforming input and output schemas, where the input schema is meant to be a container for one or more experiment points. In the following example, **a** is the input schema and **b** is the output schema.

simple_experiment$_1$: **b** = $\{$**component**$_1$**, a**$\}$

complex experiment : a directed graph of specific simple experiments defining the control-flow and data-flow in a computation.

complex_experiment$_1$: **b** = $\{$**component**$_1$**, a**$\}$; **c** = $\{$**component**$_1$**, b**$\}$

installation : the description of any computing environment available, including all the information necessary to access the environment and run components.

installation$_1$: **UNIX machine gnida.cs.vt.edu, ssh, start_process**

run : the act of instantiating schemas associated with an experiment (simple or complex). We will variously refer to a run as a specification ready for execution, in progress, or completed, depending on the context. Typically the user instantiates the input schema and the system instantiates the output schema(s). Keep in mind that both schemas must have been created by the user at experiment definition time.

run$_1$: **simple_experiment**$_1$ with **a** = $\{$**3**$\}$ and **b** = $\{\}$
run$_2$: **simple_experiment**$_1$ with **a** = $\{$**3,4,5,6**$\}$ and **b** = $\{$**6,24**$\}$
run$_3$: **complex_experiment**$_1$ with **a** = $\{$**3**$\}$, **b** = $\{$**6**$\}$, and **c** = $\{$**720**$\}$

In the above examples, notice that *run*$_1$ is ready for execution, *run*$_2$ is in progress (and has processed two experiment points), and *run*$_3$ has completed. Notice also that while *run*$_1$ and *run*$_2$ involve simple experiments (computing the factorial), *run*$_3$ involves a complex experiment (computing the factorial of a factorial).

executor : system code for mapping (simple experiment, experiment point(s)) tuples to a corresponding run that is ready for execution. If there are multiple experiment points in the input, the semantics of the executor are that the run proceeds *serially* through these points.

executor$_1$: **simple_experiment**$_1$ $\times$ **experiment_point**$_1$ $\mapsto$ **run**$_1$

Certain other concepts can be derived from the terminology above. For example, **experiment execution** is simply the act of running an executor. A **workflow** is a complex experiment together with possible transformations to its control-flow graph and data-flow graph. **Data management** can be seen as the system functions and schemas designed to organize experiment points and the result of the afferent runs. The actual presence of a component on an installation, for example the software suite MATLAB installed on the gnida.cs.vt.edu machine, can be referred to as a **component installation**. Other terms and structures will be defined throughout the thesis at the time of first usage.

2.2 What must a Scientific Workflow Management System Support?

In [8], problem solving environments are defined as computer system targeted towards solving a specific class of problems. They provide access to advanced solution methods and accommodate new ones as they become available, support the targeted class of problems without requiring specialized knowledge of the underlying computer hardware or software, and track extended problem solving tasks. Other specialized services will, of course, be required depending on the specific target class of problems. These requirements extend over all aspects of a complete workflow and data management system. There are specific issues related to each of the concepts defined above, which can be organized as follows, and discussed in subsections below:

1. Components and installation management
2. Experiment execution management
3. Data management
4. Workflow management

2.2.1 Components and Installation Management

PSE requirements for installation management primary stem from issues of component heterogeneity and accommodation concerns. A flexible system should accept components in any form presented by the user, and not be limited to a certain type such as Java programs or source code in a specific language. There are many situations where a fundamental component of the experiment is a legacy program that cannot be modified. In these cases, the component is practically static from the experimenter's view, and most often the user writes scripts to wrap around it. In other situations, especially when the components are meant for remote execution, it is infeasible to have the components installed on the workflow host system. Therefore, to achieve maximum flexibility, there should be as few restriction on the user component type as possible. Furthermore, managing user components should not require any special modifications to render the experiment realizable within the workflow management system. An operational way to think about this criterion is to ensure that the management system be component-agnostic.

2.2.2 Experiment Execution Management

To realize automated execution of scientific experiments, the workflow system must assume responsibility for transparent distribution of components and monitoring of executions in progress. Automatic distribution of user components on the computing system reduces the need for implementing a versioning system. At each instance of starting an experiment, the necessary pieces (e.g., software codes, data files) can be distributed to the participating installations, relieving the user from having to know the underlying details of such systems as clusters, remote access methods, and execution environments. In addition, if the workflow system is implemented atop a database management system (DBMS), the active elements of the DBMS [23] (e.g., triggers, rules, and event-based programming) can help automate the process of executing and monitoring complex programs.

2.2.3 Data Management

Novel design considerations emerge when we consider the data management aspects of a complete workflow system. Since the objective of scientific experiments is to transform data in a meaningful way, extreme care has to be taken to not limit the experimenter in any way, while still keeping the language of interaction simple. A common approach is to model the input and user requirements of user components (schemas) as tables in a SQL-based relational database system. Furthermore, most of the data available on the web or other sources is readily stored in relational databases; complete and accurate data models can hence be designed to accommodate many diverse information sources. Finally, coding the input and output requirements in a relational database does not limit the experimenter to a specific data type or structure. The management system should support any combination of tuple attributes that is supported by the underlying database management system – in this case, any pairing of any number of SQL data types.

Storing scientific data as tuples in a relational database also opens the discussion to partially processed (or partially generated) experiment points. The issue of partial data can be furthermore split into two categories – row-wise and column-wise. Row-wise partial data is tracked at the experiment scope. When we observe that scientific experiments are usually run on sets of inputs composed from database tuples, row-wise partial results refer to the ability of the management system to track the experiment's execution, and provide information about it in terms of the data computed thus far. In order to fully support this feature, the system must not be limited to returning merely %completion information, and must support querying the computed tuples out of the total set. This feature is most important for experiments such as parameter sweeps [3], search-based optimizations, and iterative improvement algorithms. In these contexts, the experimenter rarely desires the experiment to run for its entire duration; the execution is usually stopped when the results are deemed satisfactory or if a problem has been identified.

Column-wise partial results refer to partially computing the tuples' constituent information, while still considering the entire experiment point set. This feature is clearly different from row-wise partial results, as it omits attributes of the data points, but not the data points themselves. This form of partial results becomes very useful when considering certain cases of information retrieval applications. A user component designed to access a bioinformatics publication database such as **PubMed** might retrieve a large feature set such as the title, abstract, and body of the requested article. However if, for the purpose of the experiment, the user only requires the first two features, retrieving the body would be a waste of resources as it is typically much larger than the first two attributes combined. The management system should be configurable to allow such interpretations, to the extent of the component's support of the feature.

Another feature essential to data management is tracking the data through the chain of execution, especially in complex experiments. Having access to the complete chain of data transformations including branches, joins, and choice of conditional paths does not only ease debugging, but is also vital to interpreting experiment results. To be of utmost use, tracking must operate at the granularity of individual tuples in the experiment point set, and allow distinction between outputs even though there might be temporary mismatches between the number of input points and the number of output points. Data tracking should not be confused with experiment point caching and storing (detailed below). Many workflow management systems implement experiment point caching and claim to have both features.

Tracking experiment points introduces the concept of 'varying multiplicity' data. This term describes cases when one experiment point computes a result that actually corresponds to more than one experiment point in the range. An example of such an experiment is a 'scatter-gather' operation, popular in information retrieval. Here, the computations are staged, so that the first stage computes a multiset from a given experiment point, which could then be arbitrarily separated and re-grouped in the next stage (hence the need for modeling the range as multiple experiment points). A typical workflow solution is to explicitly program this feature but this approach still requires us to distinguish between gathering (collector) operations and regular one-to-one experiments, and is often relegated to the user's component. As one of the main objectives of the workflow management system is to hide such complexity, this feature should also be mandatory.

An oft-overlooked feature in scientific workflow management systems is eliminating the difference between already-computed data and information coming from experiments that are not yet run. In a system supporting this feature, the user could just 'query' for the experiment results regardless of whether they have been already computed. If the results are not available, the query effectively acts as a cue for requesting that the data be freshly computed or materialized. However, care must be taken when using such a feature to ensure that the target execution context supports such automatic computation in a feasible manner. Consider, for example, the following query:

```
SELECT *
FROM sentences
WHERE sentence.length > 3;
```

This hypothetical experiment is meant to be performed using a component that generates sentences of a certain length. While typical query processing engines would trap 'unsafe' queries, or re-order plans to improve efficiency of execution, it is unclear if the above query will terminate in a reasonable amount of time or constitutes an abuse of experiment specification flexibility. The fundamental problem here is that query optimization functionality does not extend into the realm of user-supplied codes; this feature hence places significant onus on careful experiment design.

Closing the discussion on data management requirements, we arrive at the issue of execution priority at experiment point level. A flexible management system should support changing the order of execution in the set of experiment points defining the input, as long as there are no dependencies between the targeted experiment points. For instance, we should be able to employ algorithms that automatically infer priorities based on knowledge about the domain and individual experiment point requirements.

Components and installation management	
	- Component heterogeneity
	- Ease of component accommodation
Experiment execution management	
	- Automated management and execution
Data management	
	- Use of relational DBMS
	- Lack of input/output restrictions
	- Row-wise partial results
	- Column-wise partial results
	- Data tracking
	- Support for 'varying multiplicity' data
	- Uniformity between computed and uncomputed data
	- Caching/storage of experiment point sets
	- Variable priority for data computation
Workflow management	
	- Experiment specification not restricted to DAGs
	- Support for domain=range
	- Adaptivity of workflow
	- DBMS as workflow management engine
	- Support for failure management
	- Stream processing
	- Change management/versioning
	- Parallelism

Table 2.1: Desirable features of a workflow management system for PSEs.

2.2.4 Workflow Management

Workflow management in a computational science context must adhere to the best practices underlying how scientists utilize codes and process results. A common approach in workflow management systems is to restrict the composition style of complex experiments to involve only directed acyclic graphs (DAGs). This however blocks the application from supporting scenarios such as optimizations or control feedback loops. Data harvesting systems [12] may also employ the use of feedback in complex experiment design. These examples and many others render the feature of supporting arbitrary execution graphs a must for PSEs. Closely related to this feature is allowing a simple experiment to have the same range as its domain. Example applications exhibiting this feature will be presented in the following chapter, but keep in mind that this feature can always be simulated if complex experiments are not constrained to DAGs. The design though does become cumbersome, as a dummy pass-through experiment must often be employed, complicating the data tracking functionality.

Probably the most important feature of a workflow management system would be its support for adaptivity. The workflow system should allow arbitrary changes in the workflow structure; how this feature is implemented should be transparent to the user. Examples of changes in the workflow are swapping components, adding or removing simple experiments to the system, or changing the path of execution. This feature is closely followed by the implementation of change management, at workflow definition level as well as execution level. Change management at the workflow definition level refers to capturing different versions of components and experiments, and labeling the data and experiment results accordingly. This feature is most often not implemented and meant to be managed by the user, through specific naming of

components. Change management at the workflow execution level is however very important, as it supports failure identification and interpretation of results.

The final advanced feature in workflow management is stream processing. Various projects (e.g., [9]) have pioneered the development data stream management systems, but such ideas are yet to be integrated into a complete workflow management system. Issues include, but are not limited, to modeling and accommodating temporal properties of the data stream, monitoring rapidly changing data, the design of continuous query languages, and query evaluation.

Table. 2.1 summarizes the various features discussed in this section.

2.3 Survey of Related Research

This section presents several systems that target implementation of a scientific experiment management system. Some were designed with a specific application in mind, thus presenting very clear design considerations and compromises. Others propose a complete framework for such a system but set forth to implement only a subset of the features. Some do provide an almost complete set of features, but fail in areas such as applicability of the workflow engine, inadequate granularity of data tracking, or restriction of complex experiments to DAGs.

2.3.1 ZOO

Zoo [1] is a project implemented at the University of Wisconsin, Madison that views scientific workflow management in terms of a web of interconnected data objects. The active links between these objects carry the necessary process descriptions that define their execution. The entire workflow is fully defined as a database schema in an object-oriented DBMS, and can describe an arbitrary experiment DAG. The focus of ZOO is proving that an object-oriented approach is the correct view of scientific workflows, and providing all the functionality of a workflow management system through the DBMS.

2.3.2 ZENTURIO

ZENTURIO [15] offers a unique approach to web-based experiment management systems for grids. While most of the systems presented in this section choose to store the workflow-related data in separate database tables, ZENTURIO places this information as annotations to the user's component source code. The system focuses mostly on experiment execution management and not on the transformations that data is subjected to during an experiment; therefore the system lacks even the most basic tracking of data. Some advanced features include adaptability to workflow changes, event based stream processing, and abstraction of the underlying grid operating environment.

2.3.3 Chimera

Chimera [7] is part of the Grid Physics Network (GriPhyN) system [2], aimed to provide grid technologies for scientific and engineering projects that must acquire and compute large distributed datasets. Chimera is the component responsible with all aspects of data management in the GriPhyN system including, but not limited to, a virtual data catalog, data derivation representation and procedures, and virtual data language interpreter to translate user queries into data definitions. Chimera incorporates features such as automatic computation of data points, full capture of data transformations, and ability to model DAG-like experiments. As one of the more complete experiment management systems, Chimera provides a good point of comparison to our system, in terms of the desirable features presented above.

2.3.4 EMDAG

EMDAG [20] is an experiment management system developed at Virginia Tech, and is also the system on which BREW builds. EMDAG provides most of BREW's experiment execution management; the components and installation management and data management are modified to fit the extended functionality that BREW introduces. While EMDAG supports very complex experiments, it does so at the expense of ease of use. Most features are given only as templates, and the responsibility of implementing the interaction between the user components and the management system is passed on to the experimenter. EMDAG also does not provide any workflow management features or advanced data tracking capabilities.

2.3.5 WASA

WASA [22] is aimed as a complete framework for computer-based environments to support scientific applications. The initial project design proposes a set of features as mandatory and advocates the use of existing database technology to create the management layer, and hiding it from the user through an object broker interface [11]. Unfortunately, in further iterations of this framework [21], the target scientific community is dropped in favor of business workflows. While the two have been shown to share some goals, the requirements for the scientific side are much more strict, rendering the new system inadequate. Nevertheless, some references to the original WASA architecture are made throughout the thesis.

2.3.6 LabFlow

LabFlow's [4] contribution is that of defining a database benchmark for high-throughput workflow management systems, together with a minimal feature set. The emphasis is placed on event history and dynamic workflows. The approach is not different from the other experiment management systems presented here; LabFlow's notions of **materials** and **steps** correspond quite closely to data points and data transformations, respectively. Being merely a benchmark, LabFlow is considered here for the guidelines it sets up for workflow management system design.

2.3.7 Telegraph

The Telegraph project at the University of California, Berkeley [5] focuses on formalizing a dynamic workflow system capable of processing streams of networked data. Technically not a workflow management system for scientific experiments (due to its focus on online query processing), Telegraph implements the complete suite of partial-result features. Furthermore it introduces the concept of reordering data streams [18] to assign user priorities to experiment data. Many of these features are not present in any other system reviewed in this paper, thus making Telegraph a very good point of comparison.

2.3.8 WSQ

WSQ/DSQ (web supported database queries/database supported web queries) [10], developed at Stanford University, is again not an experiment management system *per se*, but provides certain features worth mentioning here. In the WSQ part, the authors focus on enhancing database queries with results from Web searches. The use of a relational DBMS, coupled with the encapsulation of the extra functionality in two virtual tables indistinguishable from regular SQL tables, makes the system comparable to a component-specific management system. Other features such as support for concurrent Web search requests and the lack of distinction between already-accessed data and waiting-to-be-accessed data further increase its relevancy to the work presented in this thesis.

2.4 Discussion

From Table. 2.1 and the sample systems described above, we can make a few observations. First, workflow management systems based on an object-oriented DBMS (e.g., ZOO) typically restrict their scope to custom-designed OO components. While theoretically any type of component can be wrapped in an object-oriented interface, doing so requires programming expertise on the part of the experimenter. Similarly, even though EMDAG is designed around a relational database system, it requires experiment schema and input/output structures, as well as experiment execution code, to be painstakingly designed by the user.

Second, systems offering sophisticated functionality often require custom modifications to core elements of the underlying DBMS. For instance, WSQ/DSQ uses special query processors to achieve its goal of seamlessly integrating database queries with web searches. Other projects, such as WASA, use databases only partially—for storing program and data information—and delegate all aspects of execution and workflow management to an out-of-database component.

Finally, design decisions about some functionalities rapidly affect the feasibility of offering others. For instance, it is clear that systems that do not provide data tracking features also will be lacking in terms of correctly modeling varying degrees of data point multiplicity. This is because the latter feature is a special case that builds on the former. A careful deliberation about feature implementations is thus in order.

Chapter 3

Basic Experiment Design using BREW

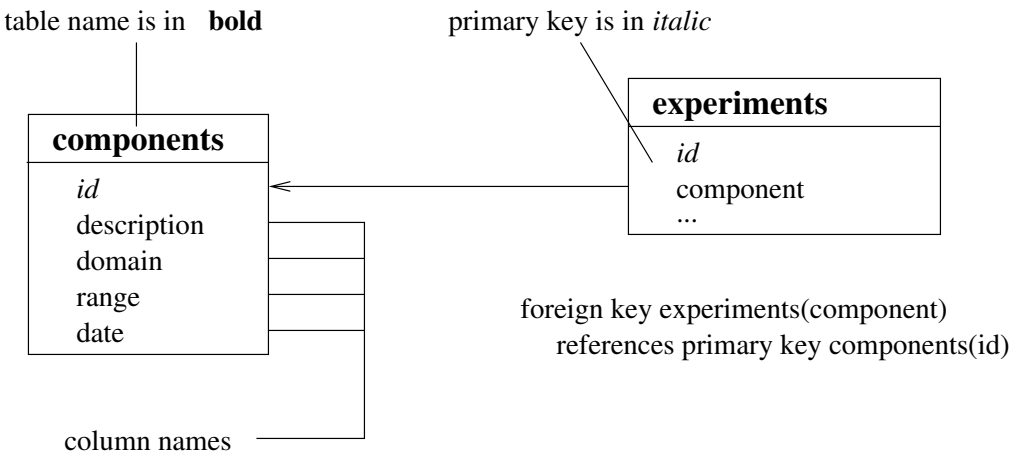
BREW is a scientific workflow management system expressly designed to address the issues raised in the previous chapter. The key design criteria of BREW are: (i) to support application-level composition and execution of scientific codes, (ii) to require minimal programming on the part of the user to setup, monitor, and control experiments, (iii) to provide as much workflow management functionality as possible from within a practical DBMS, without modifications to native software or custom software installation. This chapter introduces BREW and describes how simple scientific experiments are designed and modeled in this environment. EMDAG, the underlying experiment management system used to manage component execution, is also described in detail, along with the interaction between the two systems. The issues related to the implementation of BREW atop a DBMS (database management system) are introduced, and will be further elaborated upon in the next chapter.

In realizing the majority of its features, BREW makes use of SQL tables and queries. A moderate understanding of DBMSs is therefore assumed of the reader, although Fig. 3.1 summarizes the common conventions used in this document when describing SQL tables and modeling concepts. This model is very similar to the entity-relationship (ER) design model, with some distinctions made between certain types of relations to facilitate the understanding of large schemas. Triggers and rules are also commonly used to implement various system functionalities. All active database features and special functions are described in detail at their first reference, and complete code for the majority of them is provided in Appendix B.

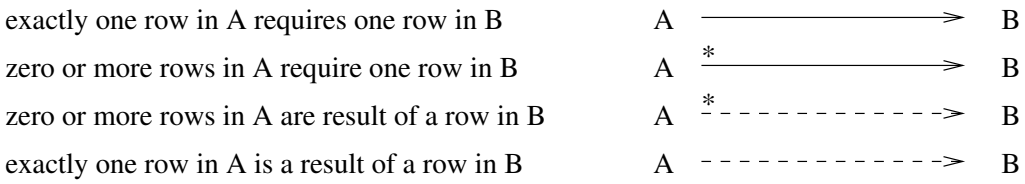
3.1 Building upon EMDAG

Recall the four basic functionalities of a scientific workflow management system from the previous chapter: BREW uses EMDAG [20], developed at Virginia Tech, as its foundation for supporting the first two functionalities. EMDAG's support for components and installation management and experiment execution management is realized via a series of database tables, triggers, and shell scripts. As a database-designed system, EMDAG (and, hence, BREW) introduces a certain amount of overhead to the simulations. Therefore it is a design requirement of both systems that the simulations be computationally intensive, to justify the additional structures.

The basic entities we will be concerned with at this stage are **experiments** and **experiment points**. Recall that through the structure of **experiments** the user wraps the simulation codes (**components**); the **experiment points** are used to wrap the simulation data, and manage it during the execution of the experiment. The following subsections expand on the idea of an **experiment**; first its basic constituents (involving components and installations management), and then how it is executed and managed (experiment execution management).



Relations between tables are implemented using foreign keys.



The different relations used

Figure 3.1: Summary of notation used in this thesis.

3.1.1 Components and Installation Management

Components are user-defined programs that are invoked by BREW to perform tasks such as generating data, transforming it, and interpreting the results. Any experiment, however complex, can be reduced to the execution of the components in a specified order on the given data. The objective of the system is to provide as much functionality to the user as possible; therefore there are no language restrictions or specialized communication interfaces. To achieve the most generality possible, the components are only required to be an external (not in database) program, such as scripts and executables. As a component, the user might have a compiled FORTRAN program that takes as input a biological model and provides a performance measure as output; another example is a program that given a URL, retrieves the corresponding HTML document, and performs some transformation.

Once components are available, one of the main functionalities of EMDAG and BREW is to support management of the components, their distribution in the computing environment, and massage all the information needed to run specific components. With distributed computing in mind, these issues of managing installations of experiments can introduce problems if left to the user. As minimum necessary, EMDAG defines three main structures: **components**, **installations** and **component installations**. Portrayed in Fig. 3.2, these structures cover the interface definition and implementation of components, definition of computing systems available for EMDAG to use, and details about specific instantiations of components on various computing systems. To define an **installation**, a user only needs to provide an environment similar to the one in which the desired experiment would run if standalone. Taking note from everyday work, the environment consists of an access method, location of data and code within the environment, and functions to

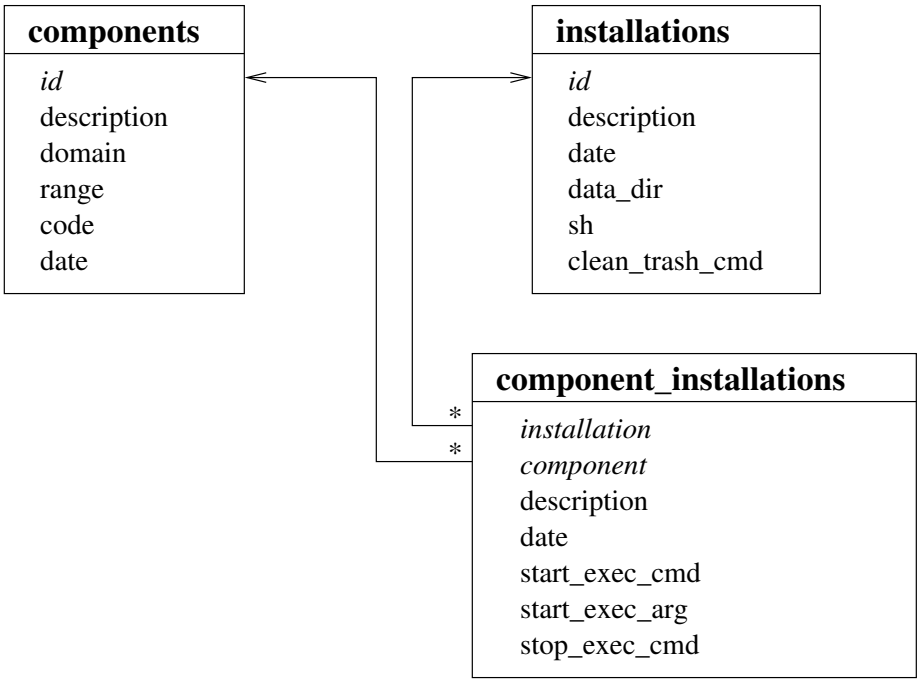


Figure 3.2: Schema for components and installations.

manage the environment, such as cleaning temporary files. In the scope of this thesis, the preferred environment is any UNIX system, primarily because of the scientific usage contexts that BREW is targeted for, and its dominant presence in computing environments such as clusters.

In the simplest case for **installations**, where the experiment is executed on the same system that holds BREW, the experimenter would define a local system by setting the access method to '`/bin/sh -c`' (or other preferred shell). This instructs BREW that in order to access the system it only needs to invoke a UNIX shell. Every system defined by an **installation** thus provides a method of connecting to it; in the case where the **installation** is not local to the database system, a simple way to run commands would be to replace the local shell with `ssh`, or any other method of running remote commands. An example of such a command is

```
ssh -l BREW -x gnida.cs.vt.edu,
```

where the name of the user is '`BREW`' and the remote computer is 'gnida.cs.vt.edu'. If a remote access method such as '`ssh`' is used, the experimenter must make sure that `known_hosts` based authentication with empty passphrase is set up, to prevent the target system from prompting for a password. This startup command is used to execute the code provided by the user in the **component** definition on the target system. Other parameters that need to be specified are the `clean_trash_cmd` command which is used to clear unused data, and the directory where data is stored. An example of the simplest installation is:

```
INSERT INTO installations (id, description, data_dir, sh, clean_trash_cmd)
VALUES (
    'default',
    'local machine where the software is installed',
    '~/data/tmp',
```

```
    '/bin/sh -c',
    ''
);
```

The code above is an example of a local installation. The `clean_trash_cmd` command is an EMDAG structure that, together with an `installation_cache` table, provides large object caching in between experiment points. The command is a provided Tcl script to remove the temporary files when they are no longer needed. The use of this feature is deprecated in BREW, since it can significantly slow down the automation of experiments. Since the automatic workflow management starts and stops simple experiments as required by the complex experiments, the cached large objects would either have to be constantly copied over when the system resumes execution or left there for the entire span of the experiment, thus invalidating the idea of caching.

A **component installation** is an instantiation of the experiment on a computer or system; all the information necessary to run the component in the specific environment is included, such as `start_exec_cmd` to start the component execution, `stop_exec_cmd` to stop it, and `start_exec_arg` for the command line arguments. The **component installations** table is transparent to the user. All the necessary data is generated by the call to `register_function` at component registration. This feature of BREW further enhances the adaptability of the workflow management system, as new components can be generated as the result of a data point computation. The component would then be automatically registered and introduced to the execution system. The user does, however, need to define the start and stop commands for the specific environment and component installation pair. This is achieved by defining two environment variables `start_proc_cmd` and `stop_proc_cmd` at experiment definition:

```
set start_proc_cmd '/home/pgsql/start_process -h localhost
                    -U testdb -D testdb -l testdb.log -o'
set stop_proc_cmd '/home/pgsql/stop_process -h localhost
                    -U testdb -D testdb -n'
```

These example commands and their SQL-based definition are specified for an installation on 'localhost' system, and are simple scripts provided to the user as part of the system. The database where BREW was installed is named 'testdb,' and it is accessed with the 'testdb' user. The system is further instructed to log the output coming from the standard error of the component program and place it in 'testdb.log'. The purposes of the `start_process` and `stop_process` scripts are self-explanatory.

The **components** themselves are easy to specify, as the important parts are the name and the code to be executed. Therefore in EMDAG, components are uniquely described through an `id` which qualifies the component name, `description`, and `date`. An addition to the original EMDAG **component** is the introduction of `domain` and `range` attributes, along with the `code`; these new attributes specify the input and output requirements of the component, and the pointer in the database to the user executable code. These additions are vital to BREW, as they are designed to aid in the automatic generation of the control structures. As EMDAG assumes that control structures are supplied by the experimenter, it doesn't particularly require the use of such features. Other projects such as ZENTURIO [15] include the information as annotation to the source file. Apart from obvious shortcomings in terms of adaptability, such an approach is also detrimental to the reuse of the user-defined components. As in EMDAG, the BREW commands to run and stop a component are scripts written in languages such as Tcl, C++, and occasionally Perl.

The tables in Fig. 3.2 show the interaction between **components**, **component installations** and **installations**. The **components** table is augmented from the EMDAG original to include the domain, range, and code. A detailed description of the domain and range tables is provided in Sec. 3.3.1. To register a **component** one must simply run the `register_function` in the database. For example,

```
SELECT register_function(component, dom, r,
               lo_import('/tmp/code'), description);
```

creates a component named 'component,' with the domain and range set to the 'dom' and 'r' relations in the database. The executable or script is loaded from the local file '/tmp/code,' and a simple explanation text 'description' is specified.

3.1.2 Experiment Execution Management

As outlined in the previous chapter, execution in BREW is organized in the form of 'runs' handled through **executors**. A run consists of the application of an experiment to one or more experiment points. A running executor processing one experiment point is the smallest granularity that the system can operate at. Keep in mind that executors proceed serially through the sets of input parameters, one at a time. If we desire parallel execution, multiple executors will have to be created, each working on one (or more) input parameter sets.

The original EMDAG system defines and makes a distinction between two execution structures – **executors** and **controllers**. The difference lies in their intended usages, and from the fact that every controller is an executor, but not every executor is a controller. The behavior of an executor is to: wait and get an uncomputed experiment point, get the inputs for the experiment point from the database, perform a computation on it using the user-specified component and installation, put the result in the database, and release the experiment point. The behavior of a controller is to: wait until a fresh experiment point is needed, create it, store the inputs in the database, wait for an executor to finish operating on it, examine the output and take a decision. However, these behaviors are just design guidelines. In EMDAG, only sample code as template scripts is provided for both the executor and controller. The user requires a detailed understanding of the experiment management system and the provided sample code in order to make use of these scripts. Furthermore, these fixed structures are implemented in Tcl, and so a significant amount of programming has to be done to adapt them to the users' requirements.

To achieve automation and ease of use, BREW takes the responsibility for designing the management structures and code away from the user, and instead automatically generates them. To achieve this functionality, the two structures above are merged into one structure, hereinafter known simply as **executor**, with the following behavior:

```
WHILE fresh experiment points left
    GRAB experiment point
    RETRIEVE experiment point data from database
    COMPUTE output for experiment point
    STORE simulation results for experiment point
    IF current experiment is a dependency for another experiment
        CREATE new experiment point in that experiment
        RUN executor for that experiment, and wait for finish
        CALL postaction function to reason about results
    RELEASE experiment point
DONE
```

Parts of this behavior are determined automatically at runtime, given the experiment definitions inserted in the database by the user. Using the experiment design guidelines presented in the next chapter (complex experiment design) the user can avoid having to write any experiment management code and intermediary structures, while still enjoying the full functionality of the system. The input and output handling is also no

Function	Description	Type
register_experiment	Insert an experiment in the system	SQL Function
start_experiment_executor	Start an executor to serially compute experiment points	SQL Function
stop_experiment_executor	Abort a running executor	Function
experiment_executors	Shows the current executors and their state	SQL Table
experiment_controllers	Shows the current executors that are also acting as controllers	SQL Table
experiment_summaries	Shows information about the registered experiments	SQL View

Table 3.1: Experiment-related management functions and their description.

longer the user's responsibility, as the required code is generated with the aid of domain and range specifications. These structures and the relevant design issues are described later in this chapter. The common functions and tables related to experiments and their state are shown in Tab. 3.1.

Failure of a running executor at any stage in the code does not represent a problem for BREW (or EMDAG). Since the executor can only work on an experiment point at a time, and the experiment point input and output are separate for each point, the failure does not propagate to other executors in the same experiment space. This model is different from ZOO, as the latter associates an experiment with an entire experiment point set. In BREW the successful release of an experiment point is deferred to the very end, after all the slave points have computed and the **postaction** function has been executed; hence any failure will be captured and recorded by the system as the failure of the entire experiment point. Unfortunately, this also means that the transaction model of the underlying database management system cannot be respected. By default behavior, a call to the `start_experiment_executor` function returns successful immediately after the executor code is invoked. Therefore the underlying experiment points might fail, but the function in which the call is made would not have knowledge of it. This behavior was adopted by EMDAG to allow for maximum flexibility in terms of parallel execution and partial results. As there is typically no connection between two experiment points in the scope of a single experiment, upon determining failure, an executor can move to the next point. The user has the option of re-examining the failed experiment points at any time in the execution, and even changing the data and marking the experiment point as available for execution (fresh). Most of the other experiment and workflow management systems do not implement such partial results. While ZOO does provide during-execution status of the experiment, no changes can be made.

Of course, the above failure behavior can be modified with the help of triggers. The insertion of an experiment point in the system can be followed by rules and triggers that automatically start an experiment executor, and enter a loop that checks the state. Upon seeing a 'computed' state the rule would succeed, whereas upon encountering a failure the insertion of the experiment point would be rejected. This behavior was found to be confusing and difficult to use for debugging features. Furthermore, two more issues surface when considering this type of insertions. First, one has to consider the fact that the execution of the experiment point might be too lengthy in terms of time for a single transaction. Second, the effects of a complex experiment cannot be rolled back the same way as that of simple inserts; the experiment point in question might generate many experiment points in a different experiment, out of which all but one could compute successfully. The succesful completion of an experiment point cannot then be rolled back, nor would it be desirable. The experimenter can use the information to determine the location of the failure, and retain the completed experiment points for cache purposes. The possible states of an executor are summarized in Tab. 3.2.

State	Description
0	Queued
1	Idle
2	Running

Table 3.2: Experiment executor states.

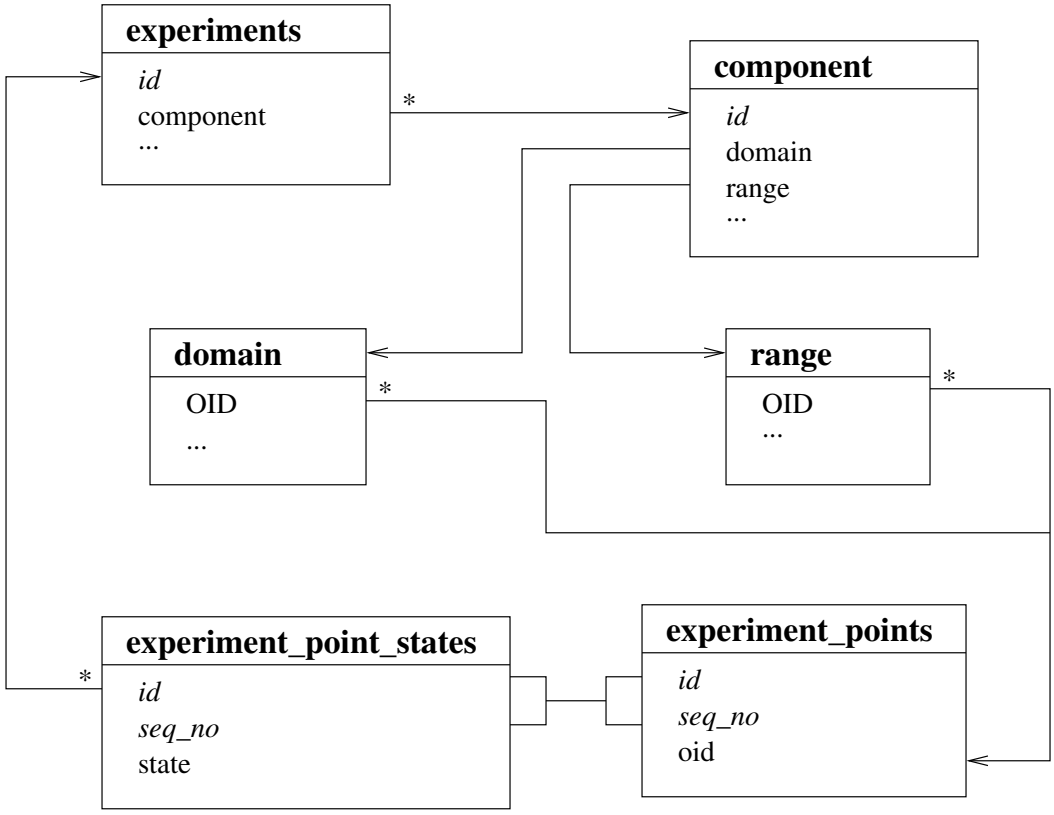


Figure 3.3: Schema for data and experiment points.

3.2 Data Management

Data management is a major part of BREW, considering how some of the more important applications of the system are information retrieval experiments. Being based on a DBMS gives the system its flexibility but does however introduce its share of restrictions that dictate the design of experiments. BREW takes a middle approach towards data management when compared to EMDAG and ZOO. The former makes no restrictions on the type of data, and only asks that it be represented by a state variable, such as the `experiment_point_states` table; the latter imposes a very strict object oriented structure that must be followed, to ensure that all data elements are normalized and thus indexable by the system. As the purpose of BREW are automation and ease of use, none of these approaches are really satisfactory. We initially adopted the EMDAG model and progressively added restrictions to fit BREW's purpose.

As seen in Fig. 3.3, BREW requires that the data be organized in `domain` and `range` SQL tables. These two tables can assume any user-defined name, as they are linked to by components and thus easily accessible. There is no design imposed on the `domain` and `range` tables, apart from the existence of the

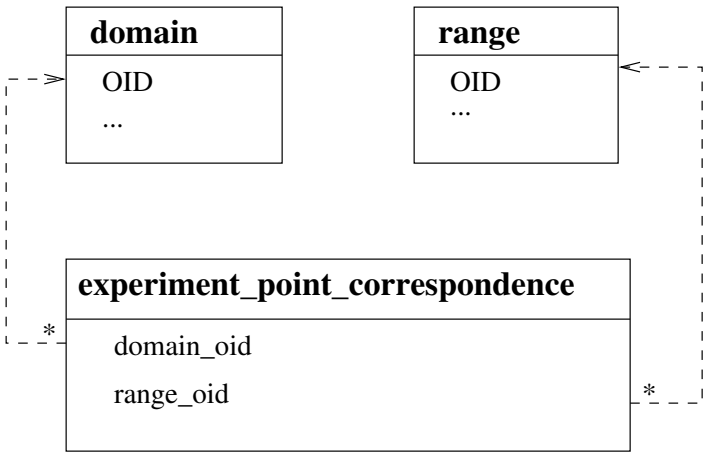


Figure 3.4: Tracking of experiment point transformations in BREW.

`OID` column. The necessity of having this attribute is described in detail in Sec. 3.3.1 below. Requiring that the experiment inputs and outputs follow the domain and range terminology is required by our goals of workflow automation; without these the automatic generation of executors and data tracking would be impossible.

The smallest data unit in BREW, like in EMDAG, is the **experiment point**. As outlined in the previous section, a component modeled by an executor can compute only one experiment point at a time. This aspect is reflected in the primary key formed out of the `ID` of the experiment and `seq_no` sequence of the experiment points. Given the unique OID associated with each experiment point, the system can easily pick and transfer the data corresponding to the particular point that is being computed. Likewise, the `OID` in the range table enables both the system and the user to differentiate easily between different experiment point outputs.

In Fig. 3.4, we see how the exact mapping between the experiment point input and output is performed. When running the component, the system automatically determines if the resulting output can adopt the same OID as the input. If true, no mapping information is entered in `experiment_point_correspondence`. If the output has to be given new OIDs (an example is given in Section 3.3.2, where we split one experiment point into many other experiment points used as input to a different experiment) then the new OIDs and the newly generated ones are tracked. This approach in no way restricts the space of experiments that can be designed and managed using BREW.

Furthermore, this system does not invalidate any of the concerns put forward by the EMDAG designers [20]. The domain of a component can be defined as a view that pulls information from other tables. The user can have any number of tables and functions unrelated to BREW; these separate entities are ignored by BREW, which only cares about the experiment input. Another difference between BREW and EMDAG is that the latter denigrates the use of triggers in data management. Through the use of views for defining domains, the user can obtain reliable automatic startup of experiments over experiment points whose data is found in multiple tables; the system can easily check if there exists a valid experiment point, or just partial data. These kinds of experiments, together with other special cases such as experiments with no input, are treated in Sec. 3.4.

Other issues put forth, such as executors finishing before all the data points are inserted in the system, are not a problem given the linking model proposed by BREW. The user can set the first experiment to automatically compute the experiment points as they are inserted in the database; if this feature is enabled, at the apparition of a new experiment point the system determines if the maximum number of executors for

Value	Description
0	Fresh Experiment Point
1	Point Executing
2	Point Computed
3	Point Crashed

Table 3.3: Experiment point states.

that experiment has been reached, and if not, it invokes another component. New experiment points created by linked experiments are also automatically processed. The concern that experiments might finish before all the slave points compute is not an issue in BREW, primarily because the target application differs. While EMDAG is oriented towards simulations, BREW is geared towards full workflow management. Therefore the system can impose dependencies created by the linking methods; as will be seen in the next chapter, the system creates a very specific tree of execution that guarantees correct order of termination.

The `experiment_point_states` table provides a quick view of the experiment points. Because BREW is implemented on top of EMDAG, it too can work with partial output. The experimenter has a very good view of what happens throughout the execution of the experiment, and can chose to stop it at any time based on the various computed points. The common experiment point states are given in Tab. 3.3.

3.3 Basic Experiment Design

As seen in Sec. 3.1.2, the smallest unit of execution in the system is the 'basic experiment.' If the system is viewed from an object-oriented perspective, the **components** can be assigned the role of class definitions, whereas the **experiments** denote object instantiation. Under these conditions, a component cannot be used by itself; it can only be used through an experiment that assigns it a specific `domain` and `range`. The execution of a basic experiment is the application of a function defined by a **component** on one or more data points. The first step in experiment design is choosing the component and its input/output objectives. The type of the component is important, as detailed below – a component can simply compute data points, generate a set of new data points from a single input, or take many inputs and reason about them. Motivation and examples of all these cases are provided in the following subsections.

3.3.1 Domain and Range Design Considerations

The user can design the domain and range for a component in a simple experiment to include any type of data in any number of columns. When the executor for the specific experiment is generated, the system automatically lays out the input to be sent to the component program and constructs the regular expressions to insert the result in the database. The rows that make up the input point are selected using the `seq_no` sequence and `OID` system identifier, and sent to the component program one by one, separated by an <–EM ROW END–> delimiter. Data from different domain columns is separated by the <–EM COLUMN END–> to allow for input and output data that contains newlines and other common delimiter characters. The experiment designer must take great care in ensuring that the specification of the component program I/O matches that modeled in the database domain and range specification. If the component program does not accept experiment points formed from more than one row, then the resulting generated system ignores the extra information, thus computing possibly incorrect results. The case when the database range is defined incorrectly is easier to debug, as either the experiment point returns empty due to mismatch of the regular expressions, or the resulting experiment point does not follow the intended clustering. To allow for as wide

Single Row:	The experiment point defined by the `seq_no` maps to exactly one row in the domain or range
Multiple Row:	The `OID` identifier is the same for two or more rows in the domain or range, causing the system to interpret all of them as a single experiment point
Single Multiple Row:	Specific only to ranges, the system interprets each complete set of output column data from the output computed by the component program as a separate range entry, even though they are produced as a result of a single experiment point in the domain.

Table 3.4: Common input/output table types.

a range of experiment points and component programs as possible, several types of domains and ranges are defined, as shown in Tab. 3.4. How they are used for experiment specification is covered in Tab. 3.5.

A `SR-SR` (`S`ingle `R`ow - `S`ingle `R`ow) experiment is closest to the mathematical function concept. In order for the executor generator to correctly interpret these points, the `OID` corresponding to the experiment point `seq_no` must be unique in the experiment's domain table. This is most easily achieved by including the "`WITH OIDS`" clause at the end of the domain table definition. While the range table could be treated in the same way and given the "`WITH OIDS`" clause, this would add an unnecessary one−to−one mapping in the `experiment_point_correspondence` table. Therefore declaring the range in these experiments as "`WITHOUT OIDS`" is the preferred design method. Keep in mind that the programmer might introduce **postactions** that could potentially alter the interpretation of 'range' for a given experiment. However, this does not affect the general BREW setup and, instead, facilitates complex linking of simple experiments that might not be easily achieved only through linking simple components. These cases will be discussed in the next chapter.

Declaring the domain of an experiment as `Multiple Row` does not make it equivalent to a function with multiple parameters. The experiment designer has to keep in mind that the columns of the domain table represent the parameters; a `Multiple Row` domain is appropriate for `collector`-style experiments. These experiments are designed to take as input a series of similar data points as if they are passed an array of elements. `Multiple Row` domains do not necessarily need to have the same number of rows for each experiment point - there are many cases when the experimenter would want the component program to be able to return varying number or rows for different experiment points.

Similar to the domain case, declaring the range of the experiment as `Multiple Row` can still be considered a simple function. However, more often than not, the real intent of these experiments is to create multiple experiment points as output, but for clustering reasons the **postaction** function is used to differentiate between individual entries. There are applications, such as similarity search that we will examine in the next chapter, where the actual intent of the experiment is to collect data for various transformations, while keeping the output from different experiment points separate. When the experiment clearly fans out one experiment point into multiple experiment points (such as splitting a sentence into its component words that are then processed individually) the experimenter can use the `Single Multiple Row` construct for declaring the range. This is equivalent to the experimenter declaring a `Multiple Row` experiment coupled with a postaction that takes the computed point, splits it into similar parts, inserts them into appropriate tables (or domains for other experiments) and keeps track of the transformation. Due to the commonality of these types of 'split' experiments and the benefits of letting BREW track the experiment points, coupled with an easier overall design and understanding of the system, the use of the `Single Multiple Row` range as opposed to postaction functions is encouraged.

Table 3.5 provides a summary of the above structures and their common pairings. The domain of the experiment is represented by 'In' whereas the range is represented by 'Out'. Experiments such as

In	Out	System OID in	User OID in	System OID out	User OID out
SR	SR	X			X
SR	MR	X			X
SR	SMR	X		X	
MR	SR		X		X
MR	MR		X		X

Table 3.5: Specifying domain and range types.

`Multiple Row - Single Multiple Row` are not included, as they can be conceptually derived from the provided domain and range types. System OID in this case represents the clause "`WITH OIDS`" to be included in the specific table; User OID is a pairing of the '`WITHOUT OIDS`" attribute of the table, and an 'oid' column of type OID in the table definition.

The effects of these definitions on data tracking is clear. In the case of `SR - SR` experiments, the attribute OID from the user-defined table is set to the system-generated OID for the computed experiment point, and no entry is added in the `experiment_point_correspondence`. The distinction between `SR - MR` and `SR - SMR` is that the range table now is given system OID in the latter, so each tuple is accompanied by a (original OID, new OID) tuple in the `experiment_point_correspondence` table. The `MR - SR` and `MR - MR` are analogous to their `SR` domain counterpart. As an experiment design rule, the first component in an experiment (or the single component, if the experiment is not a complex one that links components together) must have a `SR` domain type. Due to the automatic creation of experiment points for components, an initial `MR` domain would cause the `experiment_points` table to record an incorrect OID; even if the tuple contains some OID assigned by the user, the system will fail to identify the data for that particular experiment point, and send an empty token to the component program.

3.3.2 Examples of Simple Experiments

Consider as a simple example of domain and range an experiment which takes a text and splits it into words, as preparation for a similarity search on `http://www.google.com`. This experiment is similar to the 'term vector database' [19] in that it applies the same transformation steps. The experimenter intends to use the list of words to fetch the first page of documents returned by the search engine for each individual word, and then pick the most similar document using a standard information retrieval technique (e.g., TFIDF). Then the domain of the first part of the experiment, which takes in documents and creates words, would be defined as:

```
CREATE TABLE documents_input (
        content          VARCHAR NOT NULL
) WITH OIDS;
```

The 'content' column represents the actual document that needs to be split into words. Because the domain is an SQL table defined in the database management system, data is inserted with a simple '`INSERT`' standard query. All SQL commands and queries are available for domain and ranges, making interaction with the system abide a standard that is both easy to use and very common. The other thing to notice in this example domain definition is the '`WITH OIDS`' attribute of the table, which restricts the experiment to a `SR`-* type. Notice that this domain definition abides by the rule that all first experiments must have a `SR` domain. Given the domain, we can define the range as:

```
CREATE TABLE documents_output (
```

```
        name            VARCHAR NOT NULL,
) WITH OIDS;
```

This range is clearly the SMR type; the experiment is designed this way to facilitate the similarity search. Using a component program that takes as input free text, and outputs a list of words, the experimenter can then process the words as separate experiment points, in order to cluster the retrieved documents on words, and not put them all in an indistinguishable list (though if that's what the experimenter desires, he can achieve by just declaring this range as MR). The other pieces needed for this experiment are the register_function and register_experiment calls:

```
-- register the function
SELECT register_function('documents',
                'documents_input', 'documents_output',
                lo_import('/tmp/docs'),
                'function to generate list of words from text');

-- register a simple experiment
-- notice that postaction part is empty
SELECT register_experiment('docs', 'documents', '',
                'decompose document into words for sim search');
```

The specified code, '/tmp/docs', is a C++ program designed to read text from standard in, tokenize it, set it against a set of stop words to prune the unnecessary words, and print the words on standard output. All the code used for the sample experiments presented in the text can be found in Appendix B.

For the next experiment the assumption is that the experiment cannot yet link experiments - this will be discussed in the next chapter. Given this assumption, the experimenter desires to take the words and generate documents for them. Supposing that he just executes a SELECT statement to copy the words from documents_output to the input table of the second experiment, we can define its domain and range as:

```
CREATE TABLE words_input (
        name            VARCHAR NOT NULL,
) WITH OIDS;

CREATE TABLE words_output (
        retdoc          VARCHAR NOT NULL DEFAULT '',
        oid             oid
) WITHOUT OIDS;
```

Here the MR range type is used, as the experimenter is interested in the entire list of documents retrieved for a specific word. Of course this can be obtained also by declaring the range SR, and then executing yet another SELECT statement from the experiment_point_correspondence table; however, if the next experiment in the sequence is one that collects all the lists and generates the best document, the experimenter would have to define an additional postaction function, and take care of the mapping himself. The necessary registrations are similar to the ones presented for the previous experiment:

```
-- register the function
SELECT register_function('words',
                'words_input', 'words_output',
                 lo_import('/tmp/fetch'),
```

```
                'function to generate list of documents from word');

-- register a simple experiment
SELECT register_experiment('fetch', 'words', '',
                'gets the documents for the list of words');
```

The code for this experiment is currently a JAVA program supported by the `NetScrape` project at Virginia Tech. Details on the functionality and sample programs implemented using `NetScrape` can be found in Appendix C.

Linking these experiments, and designing a collector experiment, is beyond the scope of this section, and will be discussed in the next chapter. The last simple experiment example is in bioinformatics, and is designed to fetch from the HPRD site all the interactors for the proteins given as input:

```
CREATE TABLE hprd_in (
        name            VARCHAR NOT NULL
) WITH OIDS;

CREATE TABLE hprd_out (
        interactor      VARCHAR NOT NULL DEFAULT '',
        oid             oid
) WITHOUT OIDS;

-- register the function
SELECT register_function('hprd',
                'hprd_in', 'hprd_out',
                lo_import('/tmp/hprd'),
                'function to generate interactors for given protein');

-- register a simple experiment
SELECT register_experiment('hprd-test', 'hprd', '',
                'test HPRD for a couple of proteins');
```

3.4 Discussion of Experiment Design and Special Cases

There are some special cases of simple experiments that need to be considered. The first is an experiment with no input; in this case the component program would define an empty SQL table as the domain, and an usual range. While BREW is capable of handling these kinds of experiments, given that the execution of the system is data-driven, there would be no way to start the experiment. The user would be forced to create an entry in the `experiment_points` and `experiment_point_states` tables manually. Furthermore, data tracking in such a system will be severely reduced, as the user would have to give fictitious values for the OID, which is usually system-assigned or carried over from the parent experiment point. Another issue with these types of experiments is that their reproducibility is also affected. If the experiment generates the same results all the time, then the user should consider inserting those values directly as domain data for the slave experiments. If on the other hand the system produces different results, then the design choice would be to assign the set of generated data a meaningful identifier, in order to make sense of it later.

Another special case for the simple experiment design are experiments whose domain is composed of data coming from two or more different tables. While BREW does not support this type of experiment directly, it is only due to the automatic trigger definition for domains and ranges. Database management

systems do not support triggers and rules on views, therefore a separate set of triggers have to be defined to generate the data points. Appendix B, which presents the code for the functions used by BREW also provides templates for generating the functions. Assuming the simplest case where the experiment is based on two tables, each of which containing an OID field that is equal across parts of an unique experiment point, the user can just attach the sample triggers to the individual tables. The user can then define the domain view as the preferred selection of tables, using the common OID as the selection criteria. The extent of the triggers is to check at each insert into the partial-information table if the corresponding part is already available in the other table, and if true determine if there already is an experiment point defined for the tuple. Using this method any number of tables can be used, though the efficiency of the system drops with each new addition.

A derivative of the multi-table experiment domain can be constructed with the purpose of providing a special case of partial results. As opposed to providing missing-row partial results, the system can provide with column-missing partial results. Such behavior is seen in [17], though the system is designed specially for these purposes, and therefore does not catalog as an experiment management system, much less as a workflow management system. By removing the clause in the above solution that the trigger check the remaining parts of the experiment point, if the execution component accepts the partial input then this feature can be emulated entirely. However, unlike other systems such as [10], by not using a placeholder instead of the uncomputed data, discriminating automatically between partially-computed and fully-computed is not possible.

Other special cases are as defined in [5], and pertain to stream processing. Currently BREW does not support experiments that generate potentially an infinite amount of slave experiment points; such restrictions are imposed by the current execution method. As the component program is run with a call to the system `EXEC` command, and the output is stored and processed using regular expressions, encountering infinite output would cause BREW to time out and crash. If the stream of data was however provided to BREW by an external system, with clever component and range design the stream processing behavior could be implemented.

Chapter 4

Getting more out of BREW

Most of the advanced functionality features of BREW lie in the automatic handling of complex experiments. The basics of data management have been presented in the previous chapter; there is little to add in this category as a complex experiment can always be split into its constituent simple experiments. However, as foretold in the 'Domain and Range' section previously, there are some issues regarding linking of simple experiments; in this chapter, the basic building blocks of complex experiments will be introduced, together with a discussion of dynamic workflows. The theoretical examples will be extended by sample experiments that use the presented functionality – most will build on the simple experiments discussed previously.

4.1 Linking Experiments

In BREW, basic experiments are linked to create complex experiments either through domain and range associations or by employing postactions. The first type of linking chains components by defining the input of one experiment to be the output of the previous one in the chain of execution. All the domain and range considerations presented in the previous chapter apply fully, and the data transformation graph is tracked automatically using the `experiment_point_correspondence` structure.

Consider the part of the human protein reference database experiment presented in the previous chapter. The role of the simple component was to fetch all interactors from HPRD for a specific protein in the `hprd_in` table, and place the results in the `hprd_out` table. Suppose however that a complete index of the proteins referenced on the HPRD site is not available, and the experimenter decided to run an exploratory experiment to find out as many protein names as possible, beginning with the few she knows. Using just the basic functionality, the user would alternate steps of running the experiment on the fresh experiment points already existing in the database, selecting all the new proteins found as interactors, and inserting them in the domain of the experiment as fresh points. This approach is clearly not viable for even the smallest of experiments.

The main element of this chapter, the **postaction** function, is designed to facilitate the design of experiments that cannot be modeled solely through domain and range associations. Due to the restrictions imposed by the SQL standard, the range cannot be declared as a view, making the use of the postaction function unavoidable in certain cases.

As mentioned in the previous chapter, the **postaction** function is an SQL function defined at experiment design step, with the purpose of reasoning about the experiment point after it has been computed by the component function. The prototype that any postaction function has to obey is:

```
BOOLEAN <postaction name>(VARCHAR(32), INTEGER)
```

The first parameter to the postaction function will always be the experiment ID that generated the call, and the second parameter the `seq_no` sequence number of the specific experiment point. The actions that such a function takes are irrelevant to BREW. The postaction function can add points to an experiment, modify the execution flow of an already-existing complex experiment, add and remove components, or even create new workflows. Due to execution semantics the postaction should not be used to stop the experiment in which it is executing; it can however be used to start and stop other experiments.

The return value of the postaction is used to determine the end status of the experiment point. Even if the component program executed with no error, if the return value from the postaction is `false`, the experiment point will be marked as failed. In a way, the application of this function to an experiment point in the scope of an experiment is analogous to that of a trigger on statements such as `INSERT` in a regular SQL table.

Therefore, using the facilities of the postaction, the experimenter can extend the HPRD example to explore connected components while still storing interactions by declaring the following function:

```
CREATE OR REPLACE FUNCTION hprd_postaction(VARCHAR(32), INTEGER)
    RETURNS BOOLEAN AS '
    spi_exec "INSERT INTO hprd_in
          SELECT interactor FROM hprd_out AS o, hprd_in AS i,
                experiment_point_correspondence AS epc,
                experiment_points as EP
          WHERE ep.seq_no = '$2' AND ep.id = '$1' AND
                epc.domain_oid = ep.value AND o.oid = epc.range_oid AND
                o.interactor != i.name"
    return 1;
' LANGUAGE 'pltcl';
```

The function, written in SQL-embedded Tcl, first identifies the experiment point by which the call was generated, then picks all the interactor output data generated by the HPRD site, and inserts non-existing protein names into the input table. Notice that the output table was not used as a linking element, allowing the experimenter to use the experiment as part of a larger complex experiment.

More complex compositions are possible using both range-domain linking and use of postaction. The Workflow Management Coalition (WfMC) [6] defines several basic structures from which any composition can be modeled. Though these terms are defined in terms of business workflows, their meaning transfers to the scientific environment.

AND-Split : A point within the business process where a single thread of control splits into two or more threads which are executed in parallel within the business process, allowing multiple activities to be executed simultaneously.

OR-Split : A point within the business process where a single thread of control makes a decision upon which branch to take when encountered with multiple alternative business process branches.

AND-Join : A point in the business process where two or more parallel executing activities converge into a single common thread of control. Synchronization is needed.

OR-Join : A point within the business process where two or more alternative business process branches re-converge to a single common activity as the next step within the workflow. (As no parallel activity execution has occurred at the join point, no synchronisation is required.)

From the basic properties inherited from EMDAG, BREW supports parallelization of activities at experiment point level with no additional structures. If however the intent is to transform a single activity stored as one experiment point whose components would otherwise be executed serially into the set of constituent processes which are afterwards executed concurrently, then all that needs to be done is to define an experiment with a single or multiple row style domain and a single-multiple-row style range. The amount of parallelization can be specified as the number of executors running in the following experiment whose domain is the target range of the split action. If the resulting constituent parts are not similar, then a simple trigger can transfer them in their respective experiments, which then will process them concurrently. Convergence can be specified either through domain and range linking, or through postaction, and will be defined further below.

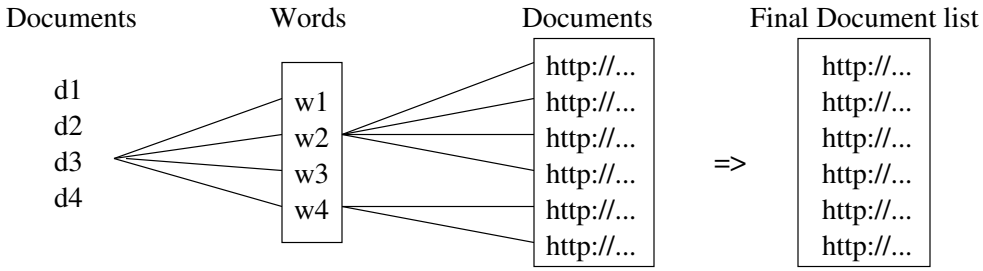
The OR-Split and any type of conditional branches can be modeled with the reasoning step of the executor, by defining a postaction function. Recall that BREW does not have an imposed specification on the form the postaction might take, to allow for as much flexibility as possible. A template for creating a boolean postaction function is provided below:

```
CREATE OR REPLACE FUNCTION if_postaction(VARCHAR(32), INTEGER)
    RETURNS BOOLEAN AS '
    # first determine the range of the experiment
    spi_exec "SELECT cp.range AS range
        FROM experiment_points ep, experiments e, components cp
        WHERE ep.seq_no = $2 and ep.id = $1 and cp.id = e.component"

    # select conditional atribute
    spi_exec "SELECT <conditional attribute>
        FROM $range"

    # check and branch
    if {<condition>} {
        spi_exec "INSERT INTO <next experiment>
            VALUES (<selection of desired attributes>)"
    } else {
        <alternative behavior>
    }
    return 1;
' LANGUAGE 'pltcl';
```

The user can decide to include the branching function directly in the experiment that it is logically bound to, or in a separate dummy experiment if the former one already defines a postaction. Notice that there is no mentioning of how and if the branches converge, as all the parameters for the convergence are separate from the split issue.

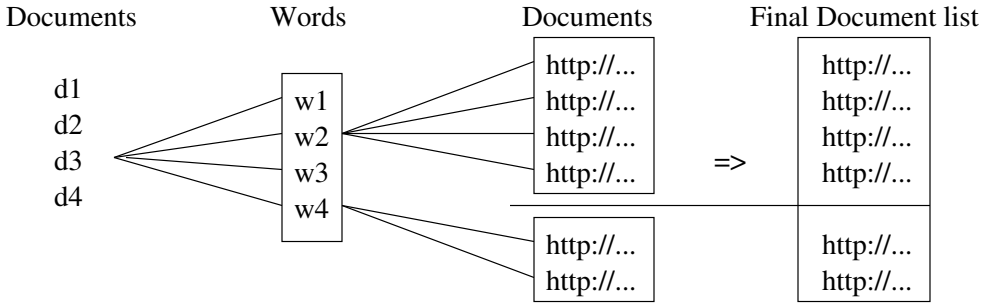
An OR-Join can be modeled very easily by ensuring that the last experiment from each separate execution branch has the range connected to the domain of the next experiment in the flow. This abstracts the conditional branches and the split point to a simple experiment, with normal input and output definitions. It is because of this abstraction that no special steps such as synchronization need to be taken.

The AND-Join most of the time can be implemented exactly as an OR-Join. The only case which is different is when the sub-tasks generated as a target for parallelization are indistinguishable. The problem arises due to the inability of the system to model specific multiplicity factors; therefore additional care has to be taken at the synchronization step.



Similarity search 1

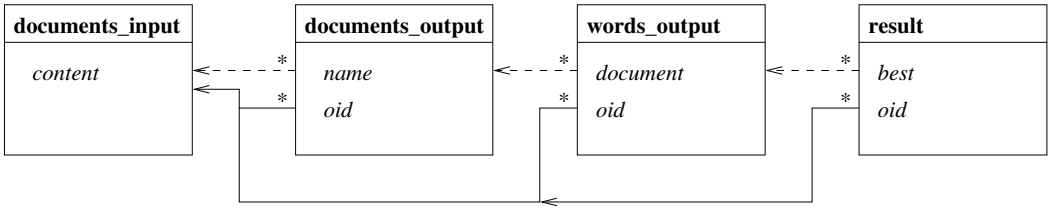
Words from same document are clustered together
Documents from the same set of words are clustered together



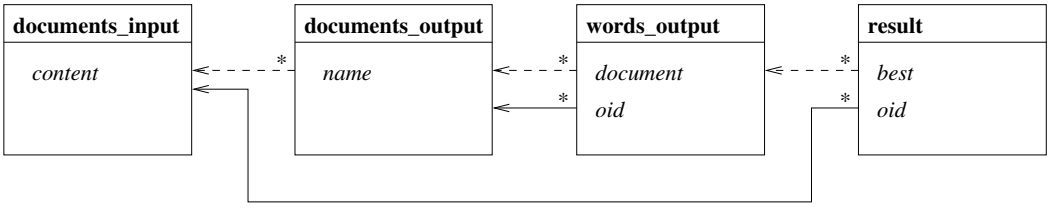
Similarity search 2

Words from the same document are still clustered together
Documents are clustered by originator word but still returned as one set

Figure 4.1: Schema for two example similarity searches.



Scenario 1: All subsequent results are bound to the originating document



Scenario 2: Documents from different words are kept separate

Figure 4.2: Schema differences for the two similarity search scenarios.

As an example of how this problem might arise, consider two variations of the similarity search for documents, as presented in Fig. 4.1. The key observation is that the first scenario does not differentiate between documents originating from different words, whereas the second does. The algorithm applied to determine the best document only requires as input a list of all the documents retrieved from all the words. In this case all the output in the system is tracked by the OID of the document that is at the root of the transformation tree. Furthermore, all the output from the simple experiment that splits a document into relevant words, which is all the words for that document, is kept in a single Multiple Row experiment point. This is presented as one input point to the simple experiment that generates the list of documents. Reusing the definition of the range for this experiment given in the previous chapter, the output from the second experiment is still Multiple Row and all the output from retrieving documents for each word is kept as one list. Similarly, the 'collector' experiment, shown in Appendix A, takes the list of documents as one experiment point and returns the most common as the best match. In this scenario all the linking was achieved only through domains and ranges.

In the second scenario, however, we cannot define the range of the experiment that splits a document into words as Multiple Row, as we are trying to make a distinction between the different words; therefore the Single Multiple Row range definition introduced in the previous chapter is used. The linking for the words→documents and documents→result experiments need not be changed, as they model the actions correctly. However, when making the distinction between words belonging to a single document, the system carries it over across all slave experiments. Therefore, the user will end up obtaining a list of most common documents, one for each word. Since an experiment point, once split into multiple experiment points, cannot be gathered back by common domain and range linking, usage of the postaction function is inevitable. Fig. 4.2, which shows the input and output schemas, clearly depicts the different clusterings performed.

Fig. 4.3 shows the basic building blocks of complex experiments. The general description of the experiment is shown alongside the equivalent structure in BREW. The latter structure is a composition of simple experiments, detailing specific domains, ranges and postaction functions. Domain and ranges are always specified, and the postaction function is omitted from experiments which do not have it defined.

Fig. 4.3 (1) shows a simple experiment, in this case with Single Row domain, Single Row range and no postaction function. Extending this simple case to model other simple experiments such as Single Row–

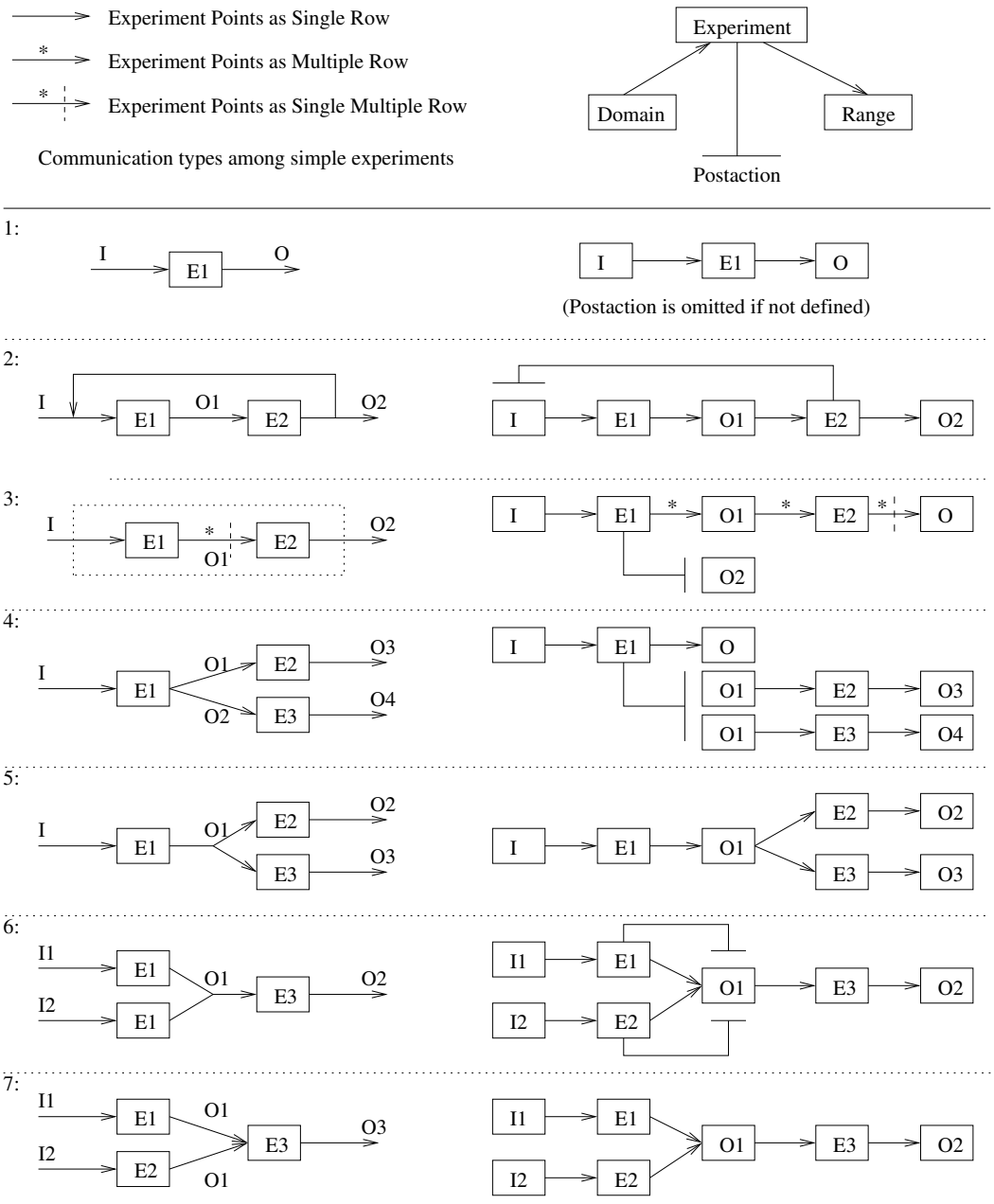


Figure 4.3: Basic compositions for defining complex experiments in BREW.

Multiple Row is just a matter of substituting for the correct arrow types. While not shown in the figure, simple experiments that have the same domain as range and carry all their actions through the postaction function are easily constructed by using the same table for both I/O structures. However, extreme care needs to be taken when designing this type of experiment, since the management system can be easily caught in an infinite executor loop. As each experiment has a limit of maximum executors that it can have running at the same time, and experiment points do not finish executing until all the slave points have finished computing, the experiment will be locked waiting for an available executor. However, if such simple experiments are needed, the experimenter can use the postaction function for feedback, using as guidelines the above described HPRD example.

Fig. 4.3 (2) is the basic case from which all complex experiments that utilize feedback are derived. This case is not different from the simple experiment feedback mentioned above, with the exception of the extra domain/range used for linking. Again, the domain and range are the simple Single-Row type, with minimum work needed to adapt it to other domain and range types.

Fig. 4.3 (3) is very useful for all cases that perform various scatter/gather operations. The range from the second experiment is not used (range O from E2) directly in the composition; the data is indirectly returned from the first experiment's postaction function, as the second experiment does not have the correct experiment point context. Furthermore, notice that only the second experiment creates a distinction between individual input tuples from the Multiple-Row experiment point, which is consistent with the practical observations arrived to at the similarity search example.

Fig. 4.3 (4) models the structure referred to by WFMC as OR-Split. The experimenter's component is used to compute the conditional variable by which the specific branch is to be chosen; the postaction function is later invoked to select the correct subsequent experiment. The template for a conditional postaction function has been presented at the beginning of this section.

The distinction between OR-Splits and AND-Splits is obvious when considering Fig. 4.3 (5). In the AND-Split case the output from the first experiment is simply used by both following experiments, no advanced postaction function needs to be called. If, because of the nature of the data, the system trigger to automatically generate `seq_no` sequence numbers cannot be used to update both experiments, a dummy table can be employed which implements a trigger that simply inserts the data sequentially in each experiment. The user can also choose to achieve this effect with the help of the postaction function.

Fig. 4.3 (6) and Fig. 4.3 (7) correspond to AND-Join and OR-Join respectively. The use of postaction in the former case and not the latter reflects the synchronization needs of the AND structures. These structures are analogous to the special cases presented at the end of the previous chapter that describe domains formed from several tables and support for column-wise partial results.

4.2 Failures in Complex Experiments

One of the first issues that appear when discussing complex experiments pertains to transactional semantics. Much like in the simple experiment case, rollback is difficult if not impossible to define on the targeted complex experiment, and might not even be desirable. When the postaction function fails for a specific experiment point, the slave experiment points or the computed data are not removed from the range of the function. The reason behind this decision is that the component program did compute succesfully, and so did any slave experiment points the controller part of the executor might have created. The failure of the postaction is reflected however in the status of the experiment point and that of the system; the former is marked as failed, while a message is generated in the `experiment_point_diagnostics` system table announcing the reason of the failure. This case is rarely a concern, as the component program and postactions should be designed to handle all input presented to them; the presented behavior should only be used to debug experiments still in development phase.

Another problem arises from the impossibility of specifying multiplicity values in parent-slave relations. By tracking the data transformation tree modeled by the system `experiment_point_correspondence` table, and through design of the component and domain and range requirements, the experimenter can specify *only* that varying multiplicity is allowed. If for example the component program modeled by the target simple experiment is required to generate exactly 10 results for each input experiment point, such a check should be performed by the postaction function. In cases where the data is of a simple type (can be represented by an SQL data type such as INTEGER or VARCHAR) and the multiplicity to be modeled is low, this fixed behavior can be modeled by changing the range definition to a complex data type formed by as many columns as the desired expansion factor.

Adaptive workflows also present some problems when multiple executors are allowed on the simple experiments that participates in the change. As an experiment cannot stop itself, the entity performing the change has to be external to the complex experiment that initiates it. The result of this restriction is not necessarily bad, as the user can model this by creating a 'change' dummy experiment. Specific 'change' experiments would need to be created at each point in the workflow where adaptability is an issue; this however implies complete knowledge of all possible workflows for a complex experiment at experiment design point, and is difficult to track. Several possible approaches are presented in Chapter 6 as future work.

4.3 Some Caveats

One of the first issues that arise when using a DBMS as the execution engine is the issue of direct interaction with the user. No matter how the component code/executable is wrapped around by the workflow management system, once in execution the component can only receive static input from data stored in the database. This limitation is imposed by the traditional mode of interaction with database management systems, in which data can only be exchanged through `INSERT`, `SELECT`, `UPDATE` and `DELETE` statements. While theoretically the experiment could be specified in such a way that upon component initialization, a direct connection with a terminal is established, the practical use of such a component is very little. More issues with such a system arise when considering that an experiment can have multiple executors running simultaneously, and that more than one component can therefore request user interaction at the same time. Solutions for this problem are left as future work.

An indirect side-effect to allowing the computing environment to be different from the one running BREW is reflected in cases which require the user component to use a different mode of interaction with the management system. The default I/O system of passing the data as text on standard input and collecting results from standard output is not adequate for cases where binary I/O is required, nor when the experiment point data is very large. While both cases can be resolved by encoding the binary data in ASCII format using readily available system utilities, or increasing the terminal limit for the executor's terminal wrapper program, none are desirable as long-term solutions. The component program is normally not given information of the management system such as database URL, and access information such as username and password; having the user component aware of such data would break design principles described in Chapter 5. Storing the specific input or output as files relative to the computing environment in which the user component was executed is only a partial solution; even if the path to the results is considered as a pointer and included in the output table, the creator of the experiment might not be able to access it due to security privileges. As component programs are started from within the DBMS, the creator of the component's process is usually the system user who runs the database process. In most UNIX installations using the Postgres DBMS, the system user is 'postgres' or 'psql.' If such alternative communication channels are used, the administrator of the computational resource must assure that users of BREW have access to the data directory defined by the system's specific entry in the `installations` table.

Chapter 5

Application Case Studies

To exercise BREW we present two case studies that involve both computationally intensive components and complex data flows through large information sites. We primarily concentrate on bioinformatics applications as they present rich opportunities for experiment management.

5.1 Modeling the Transcriptional Regulatory Network of *S. Cerevisiae*

An important problem in bioinformatics is understanding the regulatory mechanisms underlying gene expression, i.e., which gene is expressed under what conditions and how its expression is regulated (by 'transcription factors' encoded by other genes). Our first case study involves modeling the transcriptional regulatory network of the budding yeast *Saccharomyces Cerevisiae*, a model organism for Eukaryotic systems. The goal of the study is to produce a directed graph between ORFs (open reading frames, or genes) where an edge exists between two entities if the transcription of one is regulated by the other. To model this network, we primarily harvest data from the public domain SCPD database hosted by Cold Spring Harbor Lab. The NetScrape batch browser (see Appendix C) was utilized to provide basic component functionality for web modeling and data extraction.

The data to be processed consists of 104 transcription factors, each with a varying number of regulated genes. We first obtain a set of transcription factors from the top-level SCPD site and parse the resulting table that contains the gene names. To obtain the regulated genes for each transcription factor, the experimenter has to click on the name of the particular factor, and furthermore click the 'Get regulated genes' button part of the topmost form. A new page is presented, containing the list of genes. In the SCPD database, the 104 transcription factors collectively regulate the expression of 472 genes.

5.1.1 SCPD Components and Experiment Design

This experiment was implemented using 4 components, which were selected to approximate to the highest degree the actions a user might undergo in finding all the elements, browsing by hand or using shell scripts. The four components are:

get_factors : JAVA program written using NetScrape, which takes as input a URL and parses the transcription factor names. The list is printed on the standard output with one transcription factor on each line, as a result of receiving the input URL on the standard input. This component accepts only one input element, ignoring the rest if they exist.

get_regulated : Another NetScrape JAVA program that takes as input transcription factors and retrieves the list of regulated elements. As there can be more than one regulated element for certain transcription

factors, the program can output one or more tuples for each line of input. The list of tuples is printed on the standard output, with two lines for each element: the first line contains the specific factor and the second line contains a regulated element. This component can receive as input any number of lines representing individual transcription factors.

conn_components : A C++ program that takes as input arbitrary undirected graphs and generates specific connected components. The program takes as input any number of lines describing the edges of the graph. Each edge is represented by two lines, one for each participant node. Output of the component is a list of connected components formed by tuples of lines; each tuple represents the name of the connected component and the description of the file containing `GraphViz` interpretable code for the graph. The name of the component is generated automatically relative to the other components in the input graph; the file description is simply the path of the code file in the installation.

plot : A bash script together with the GraphViz `neato` component, that generate a PostScript image file for a given graph description. The bash script that formats the input and launches the `neato` component takes as input the name of the connected component and the location of the code file. As output the script returns simply the path to the image file in the installation's filesystem. This component takes as parameter only one graph.

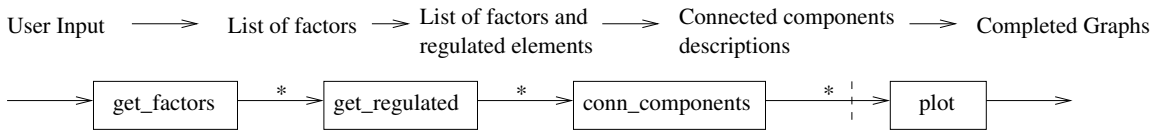
Notice that the components correspond to typical steps an experimenter might take if this experiment was designed as a suite of sequentially executing programs, where input-output communication is effected through files or pipes. As described in the previous chapters, the components are language-agnostic but adhere to standard in/standard out communication constraints. The last component is an example of wrapping a user program that does not conform to those constraints.

The tools provided by Graphviz were chosen for ease of use; furthermore they provide an acceptable layout for the final graph, and can output to a large variety of file types. The 'spring' algorithm used does create problems as it is very difficult to avoid overlapping nodes, and most importantly edges crossing the area represented by a node. However, there are no other suitable methods for laying out large computer generated graphs; the speed of execution was also considered as a factor, given the exploratory tasks this experiment was designed for.

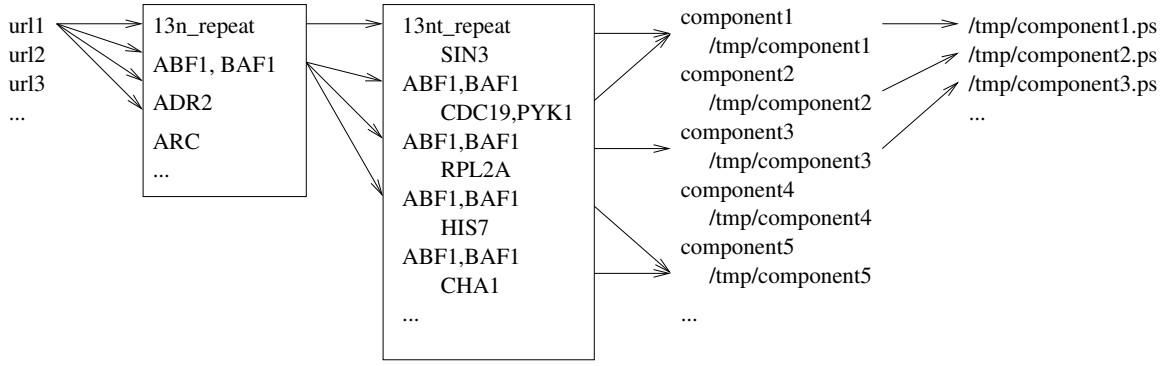
Figure 5.1 shows all aspects of the SCPD complex experiment. The first view shows the interaction of the user components described above. The experiment notations introduced in Fig. 4.3 are used to capture the domain and range requirements of the experiment. Notice that apart from these constraints, there are no additional elements needed to specify the experiment; its representation in BREW is almost identical to representing it as a series of shell commands.

The second part of Figure 5.1 uses parts of a specific run as an example to further clarify the data transformations that occur during the execution of the experiment. The design decision of passing the data as a Multiple Row experiment point up to the step that computes the connected components extremely simplifies the experiment. Had the experimenter chosen to define the connection between URLs and factors as a Single Multiple Row, the need of a postaction function would have been inevitable. Clustering all the regulatory elements and transcription factors as one list specific to each URL is also made possible by including a reference to each specific name in its corresponding list of regulated genes. The experiment is simplified even more since, in order to determine the connected components, the user program needs a VIEW over all the regulated genes over all transcription factors.

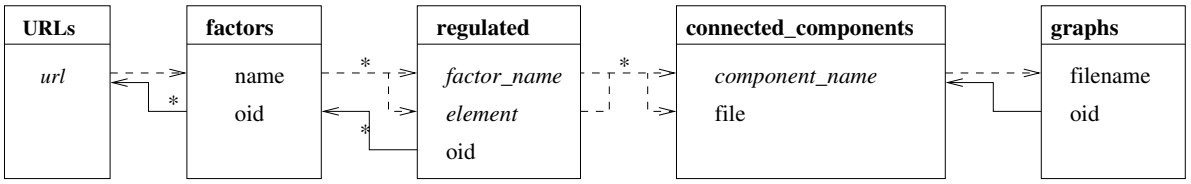
The final part of Figure 5.1 provides the specific structures for the domains and ranges used by the various simple components. The change in the multiplicity of data between the regulated genes and connected components is reflected by the break in the OID chain used to maintain the reference to the URL. This reference is however still reflected in the `experiment_point_correspondence table` (not shown here).



Overall schema of the experiment, with an emphasis on component interaction



Schema of the data transformations



Domain and Range details

Figure 5.1: Design schemas for the SCPD experiment.

When run on the SCPD site, the experiment produces 13 connected component graphs, the largest of which is presented in Fig. 5.2. Notice that this figure is plotted without directionality of arrows, owing to limitations of the graph layout software. A smaller connected component is presented in Fig. 5.3, this time using a directed graph that describes the direction of the regulation. Plotting the distribution of the outdegree of the transcription factors, we obtain the power-law distribution presented in Figure 5.4. Calculating the distribution of outdegree can be obtained by running a simple `SELECT` SQL statement; alternatively, the user can design another experiment in the system, linked to the main one through the use of postaction. This postaction would be connected to the 'get_regulated' component, as it would take advantage of the clustering of all the transcription factors.

Studying the structure of transcriptional regulatory networks in this manner is an important endeavor in computational biology [13]. For instance, the power-law behavior of the degree distribution has important implications for how cellular responses to external (as well as internal) signals are mediated and propagated. Similarly, structural components of the transcriptional network (e.g., weak ties as observed in Fig. 5.2) provide insights into the robustness properties of the cellular machinery underlying regulation.

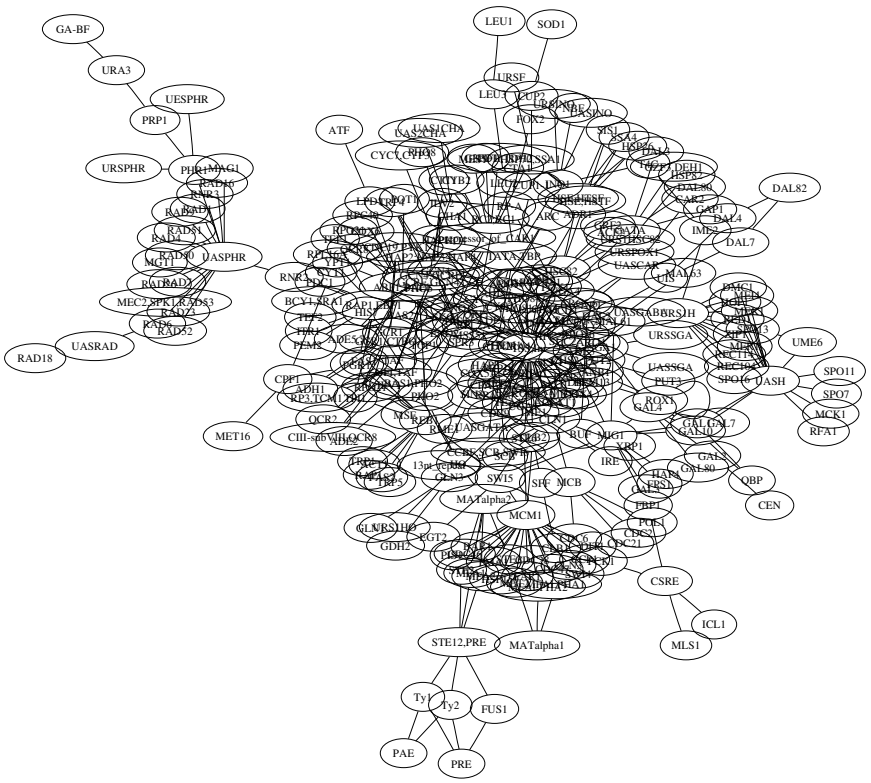


Figure 5.2: Largest connected component graph generated through BREW.

5.1.2 Performance Measurements

In a traditional shell scripting model, the experimenter would design the SCPD experiment as a set of component programs, supported by flat files that store intermediary data. This however restricts the components to running *serially* in the scope of the experiment and acquires large performance penalties due to connection setup time, network latencies, and component startup time. Furthermore, if the experimenter desires to run two experiments at the same time on different URLs, separate instances of the same experiment must be created in order to avoid mixing the results. While these issues can all be avoided with the use of pipes and different output redirections, the experimenter loses the convenient method of storing the intermediate results as files. Other specialized components can be employed to address this problem, but doing so increases

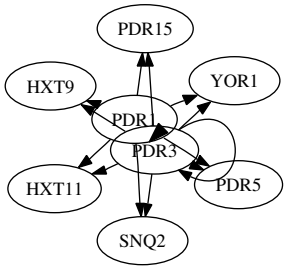


Figure 5.3: Smaller connected component together with the relation between the elements.

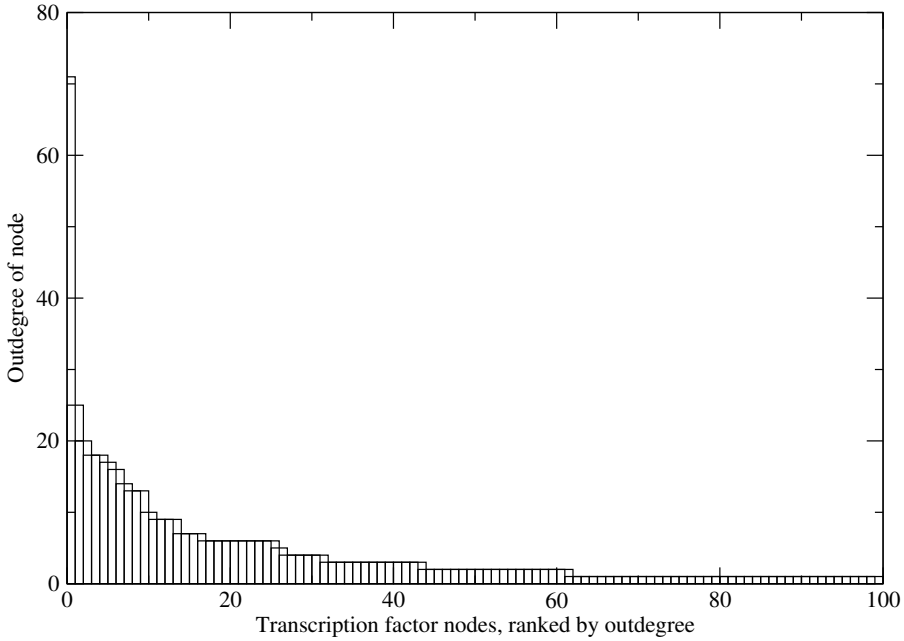


Figure 5.4: Plot of the outdegree of transcription factors, arranged in decreasing order.

C1 Invocations	C2 Invocations	Shell C1 Design	Shell C2 Design	BREW C1 Design	Brew C2 Design	Shell Time	BREW Time
1	1	All records	All records	1 ep, 1 exec	1ep, 1 exec	90s	90s
1	104	All records	one at a time	1 ep, 1 exec	104ep, 1 exec	120s	140s
1	104	-	-	1 ep, 1 exec	104ep, 2 exec, serial	-	146s
1	104	-	-	1 ep, 1 exec*	104ep, 2 exec	-	100s
1	104	-	-	1 ep, 1 exec*	104ep, 3 exec	-	88s
1	104	-	-	1 ep, 1 exec*	104ep, 4 exec	-	83s
1	104	-	-	1 ep, 1 exec*	104ep, 5 exec	-	80s
1	104	-	-	1 ep, 1 exec*	104ep, 6 exec	-	79s
1	104	-	-	1 ep, 1 exec*	104ep, 7 exec	-	78s
1	104	-	-	1 ep, 1 exec*	104ep, 8 exec	-	79s
1	104	-	-	1 ep, 1 exec*	104ep, 9 exec	-	80s
1	104	-	-	1 ep, 1 exec*	104ep, 10 exec	-	81s

Table 5.1: Summary of the data from performance testing of the SCPD application. Each row specifies a comparison between a shell script configuration and a BREW configuration. The rows marked (*) indicate when component C1 was modified to allow concurrent execution of component C2.

the experiment's complexity.

The same experiment, designed in BREW, does not present any of these concerns, as all the management is performed in the database. There is however a certain level of performance degradation, due to the increase in the number of operations that have to be performed to complete an experiment. As will be seen below, these losses amount to rendering the BREW-supported experiment slower than the shell-based only in the worst case. However, the worst case is specifically designed to not take advantage of any of the advanced features that BREW provides, and is included only as a measure of the overhead of the system.

To thoroughly assess the performance of the system, several comparisons were designed and run, both as shell scripts and as experiments in BREW, by modifying the behavior of the data collection components. The time summaries (in seconds) are presented in Table 5.1; For ease of understanding, the `get_factors` component is denoted as 'C1' (for Component 1) and the `get_regulated` component is denoted as 'C2' (for Component 2). Executors are abbreviated as 'exec' and experiment points as 'ep.'

To understand what Table 5.1 means, let us look at its first row. Here, the shell script experiment is designed to involve only one invocation of each component, i.e., after it receives the 104 records from C1,

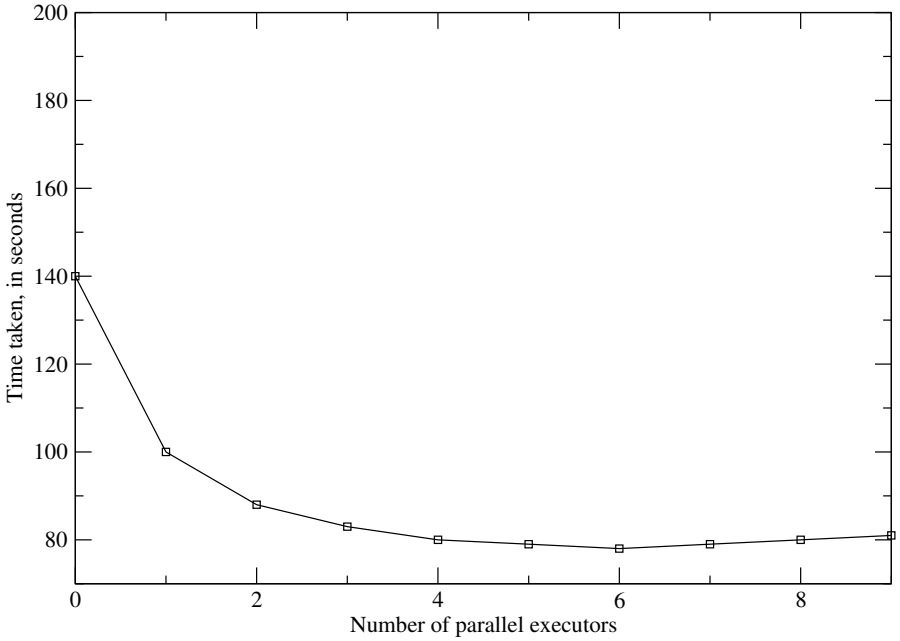


Figure 5.5: Performance of the system given different number of executors.

the experiment invokes C2 using this entire list of transcription factors. This experiment is compared with the BREW design where there is one executor with one multi-row experiment point. The results are identical in both cases (90s), meaning the overhead of utilizing the database for data communication is negligible.

The second comparison in Table 5.1 is similar to the first except that the output from C1 is split into individual records, each of which is processed by a *separate instance* of the C2 component. Similarly, the BREW design utilizes one executor in a Single Multiple Row experiment. Here, the overhead introduced by the database is non-neglgible. From the shell part of the test it can be inferred that the total component startup time is 30 seconds, which means that the database overhead is on the order of only 20 seconds, or 14 percent.

The overhead of an idle executor can be seen in the next comparison, where there are two executors in BREW but they are made to proceed serially because C1 does not have parallel functionality.

At this point, C1 was changed to include a post-action function that inserts experiment points into the domain of C2, without blocking for C2 to complete. The rest of the comparisons thus do not have an equivalent in the shell model, as there is no easy way to parallelize a shell script. However, the speedup is observed instantly, even if the management system is running on a single-processor computer and the component startup time is factored in. A plot of the time taken for the experiment to execute against the number of executors is presented in Fig. 5.5.

Since a significant part of the experiment runs over the network, there are several considerations related to testing. To factor out as much as possible random network delays, fluctuations in traffic load, and load of the web server used as the data source, the experiments were run at various times of day, and from different locations on the network, with the majority of the runs being performed at night. As little computation is involved on the server side of this experiment, there are no issues of cached results on the server; however, there could be caching effects due to network proxies. To overcome any potential problems, computers both from within and outside of Virginia Tech's network were used.

Furthermore, the server does little more than database lookup and page generation, which are not a significant source of time penalties in the system. Coupled with the fact that web servers are specially

designed to handle multiple concurrent connections, performance measurements can be run even if there is just one logical component on the server side. However, increasing the number of executors above a certain threshold does no longer improve the results. On the contrary, the performance of the single component becomes a limiting factor, and the overhead of the system begins to increase beyond the benefits of more executors.

5.2 Distributional Analysis of TonE Elements in the Human Genome

For our second study, we aimed to conduct a distributional analysis of the tonicity response element (TonE) across the entire human genome. The TonE element is a regulatory construct found in the upstream regions of genes (promoters) and is involved in response to diverse conditions that pertain to salt and water-deficit stress. By studying the relative frequencies with which these elements are distributed, we can formulate important hypotheses about the relationship between humans and primitive organisms where the TonE elements are more prevalent.

This case study is different from the previous one presented in this thesis. Instead of the experiment being designed from many components with complicated data transformation chain but that execute fairly quickly, this study takes advantage of BREW's capabilities to run computationally intensive software.

5.2.1 Promoter Analysis Components and Experiment Design

The primary computational component is the `upstream` script written in the Perl programming language. As input, it processes chromosomes (in the form of GenBank files) and a list of motifs (regular expressions), and outputs a list of recognized promoter elements in the genes' upstream regions. The script parses the given chromosome file completely into memory, building a hashtable of all the genes. After constructing this table, the script finds the elements of interest in the promoter region of each of the genes found. To take advantage of the pre-processing performed by this script, the user is encouraged to present as many motifs as possible for each given chromosome, since significant time is spent in parsing the file and building the initial hashtable. As configuration parameters the script accepts values for the maximum number of elements to search upstream, and the least number of elements to search for before the start of a gene. The default values of 1000 for both these parameters are used in the test runs described here.

The original system in which the `upstream` component is used is a suite of `Makefile` scripts written for UNIX compatible operating systems. The scripts organize the input parameters to the `upstream` component and ensure that the dependencies for the chromosome files are resolved before execution. This Makefile-based system also provides crude organization of output data; based on the functionality of `make`, already computed output files will not be rewritten. Also, based on the availability of two or more processors in the computing system, the scripts can use the `-j` option to run several chromosomes in parallel. Table 5.2 shows the time taken to process each chromosome of the human genome in minutes, its file size in MB, and the number of promoter regions returned as output. All results are relative to finding TonE elements, and have been obtained on a single processor Intel Pentium machine running at 2.4 Ghz.

The complex experiment template which fits this case study is Fig. 4.3 (5). The experiment is similar to an AND-Split since an experiment point is run in parallel by one or more experiments (see Fig. 5.6). As there is no processing step that must be performed, the user inserts the data directly in the shared domain. This requires the modification of the basic trigger function, to create the necessary experiment point structures in all experiments linked to the domain. While the system functions could be used to generate such a trigger, a specialized function (whose code is shown in Appendix A.4) is used, in order to take advantage of the optimizations provided by the `upstream` script. Notice that even if the `motifs_table` is the first domain in the system, it does not have a system assigned OID. This allows the user to create Multiple Row

Chromosome	Filesize	Time to Complete (min)	Number of Promoter Elements Found
1	400	16	62
2	390	33	41
3	326	35	39
4	309	14	42
5	296	12	45
6	301	21	42
7	519	43	78
8	240	9	30
9	213	11	22
10	237	11	31
11	236	15	46
12	228	11	30
13	168	11	18
14	146	16	28
15	138	8	21
16	141	6	16
17	136	7	23
18	126	4	12
19	98	8	38
20	116	6	15
21	62	3	3
22	68	4	15
X	239	7	20
Y	36	44 sec	0

Table 5.2: Summary of results from TonE promoter analysis along the human genome.

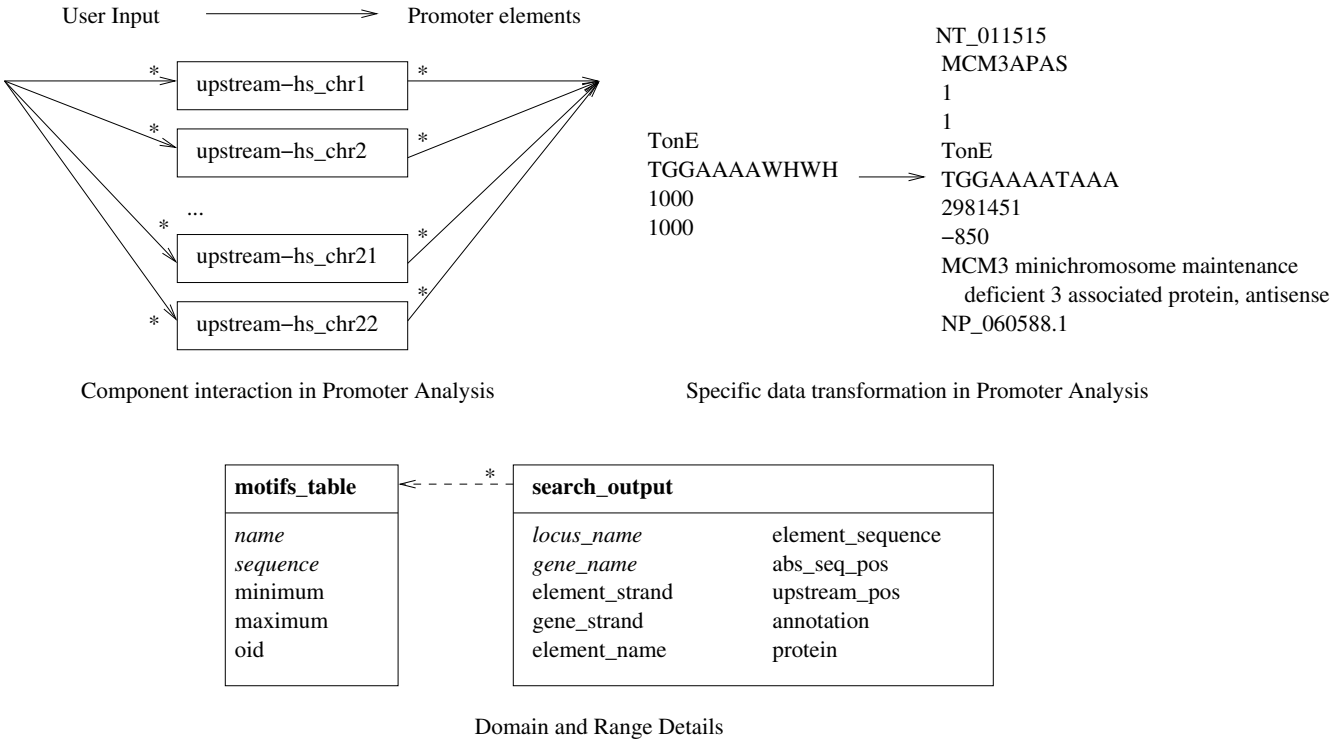


Figure 5.6: Design schemas for the promoter analysis experiment.

experiment points by hand, so that all motifs present at the start of a run are processed at once. As each output tuple contains all the necesary information to be uniquely identified, the range of the experiment is set as Single Row (or Single Multiple Row) through the use of system-assigned OID values.

The promoter analysis experiment runs two loops for each motif. The first loop is meant to process all the chromosomes, and the second is used to process all the genes in each specific chromosome. As the user component script optimizes the inner loop for each specific chromosome by building indexes that it reuses for multiple motifs, the experimenter can use BREW to parallelize the code for different chromosomes. By preparing specific component installations on various systems and splitting the list of total chromosomes among those installations, the experimenter can effortlessly improve the performance of the experiment without sacrificing ease of use and reduced complexity.

While the experimenter can provide this functionality manually, it is not desired for large-scale experiments. Running the component manually is not only prone to errors but also complicates versioning and general organization of the data. The results are not collected automatically, and distributing a new experiment must be individually taken care of for each component. Furthermore, the experimenter cannot easily set batch jobs where new lists of motifs are queued up while the current one is being processed; such functionality could only be achieved through specialized systems programming.

5.2.2 Performance Measurements

Running the `upstream` script serially on each chromosome for a given list of motifs takes the same time for both a shell-script based implementation and for a single BREW component. Assuming that all the necessary GenBank files are present on the computing resource at the time of invocation, the overhead for transmitting the motif list and other parameters to the component and instantiating various management

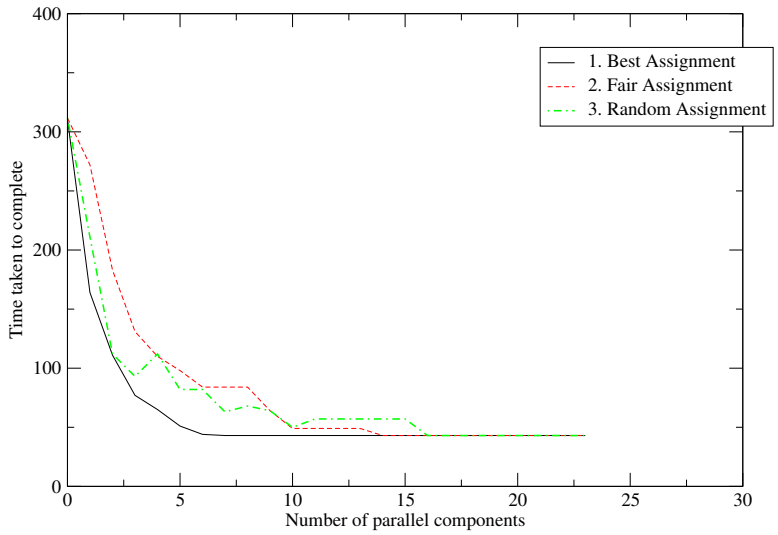


Figure 5.7: Performance of various assignment schemes for the promoter analysis experiment.

structures is 8 seconds for a local computing resource, independent of the chromosome file used. The overhead for remote resources varies as a function of the network connection linking the system hosting BREW and the computational system.

When splitting the list of chromosomes, the individual elements can be assigned to computational resources either with prior knowledge of the approximate time taken to process each individual chromosome, in a structured way with no assumptions about the data, or randomly.

Assigning chromosomes with prior knowledge of the processing time yields the best values for the system. Here, for a given number of executors, the chromosomes are equi-partitioned using a set cover algorithm that aims to balance the total expected time for processing the assigned set of chromosomes (using the data from Table. 5.2). However, since the computation of a list of motifs on one chromosome can be seen as an atomic operation, using more components in parallel does not yield better performance. As seen in Fig. 5.7 (1), using more than 7 components does not decrease the time taken to obtain full results. However, the experimenter does have access to partial results returned from faster-executing components.

Splitting the list of chromosomes equally among components, with no prior knowledge of the time taken by each individual one to compute, produces the graph in Fig. 5.7 (2). While this method does not produce results as good as the method with prior knowledge, it maintains the trend of the best curve. Most importantly, it does not make any assumptions about the data, so the experimenter can eliminate the step in which all chromosomes have a complete run. Keep in mind however that, as noticed in Table 5.2, the time taken per chromosome tends to decrease as the number of the chromosome increases, therefore still providing a ordering to the list.

Fig. 5.7 (3) shows a possible graph for a run similar to the previous run, with the specific chromosomes being assigned randomly, while obeying the balanced assignment of number of chromosomes per component. The curve is not strictly decreasing because of the synchronization step which does not consider an experiment to be finished until all the chromosomes have been processed.

Chapter 6

Discussion

We have presented BREW, a complete scientific workflow management system designed around a relational DBMS and targeted at many computational science contexts. By centrally situating the workflow management around a DBMS, the cumbersome process of defining, managing, monitoring scientific experiments is mapped to the task of configuring database tables and working with structured tuples. The two case studies have demonstrated the applicability of the system for important bioinformatics problems.

6.1 Comparison of Scientific Workflow Management Systems

We have presented a complete description of the BREW environment; comparisons with the other systems described in Chapter 2 are now in order. Many systems studied there are designed for a specific class of applications. An example is ZENTURIO, which is aimed specifically for grid environments and focuses on a web-service view of components. Some systems, such as WSQ/DSQ and TELEGRAPH, do not implement a feature set large enough to be considered workflow management systems; however, they do present some of the advanced features mentioned in BREW, such as stream processing, partial result capabilities, and heterogenous components. WASA and LabFlow systems are both presented as frameworks. WASA proposes a necessary set of features for any workflow-supported scientific experiment system, though it is heavily influenced by business workflows. LabFlow approaches the issue by defining critical features from a benchmarking point of view.

Table 6.1 shows a comparison of the mentioned systems and the features they implement. At the intersection of a system and a feature, the table can have four values. A 'Yes' indicates that the system fully supports the feature, a blank indicates lack of support, and a '*' denotes that the feature is only partially supported or some equivalent of it is presented instead. A value of 'N/A' indicates that the specific feature does not make sense in the context of the system — either the target application is such that the feature does not apply, or the design decisions imposed by the system are mutually exclusive with the feature.

6.2 Future Work

The current version of BREW does lack complete support for certain features mentioned in Chapter 2. The first such feature is change management and versioning. Neither change at workflow definition level nor at workflow execution level are fully captured.

Change management at the workflow definition layer is easy to implement. The construction of a formal library of components with annotated name, version, and description, would be followed by the inclusion of the specific version used in computation for each output set produced. Simple queries should also be provided to interact with the versioning system; querying results could be restricted to displaying data only

Feature	WSQ	Telegraph	Chimera	WASA	ZOO	EMDAG	ZENTURIO	LabFlow	BREW
Components and installation management									
Component Heterogeneity		*	Yes			Yes			Yes
Ease of component accommodation	N/A	N/A	Yes	*		Yes			Yes
Experiment execution management									
Automatic management and execution	N/A				Yes	Yes			Yes
Data management									
Use of relational DBMS	Yes	Yes	Yes			Yes			Yes
Lack of input/ouptut restrictions	N/A	*	*		*	Yes			Yes
Row-wise partial results	*	Yes				Yes			Yes
Column-wise partial results	*	Yes		*			N/A		*
Data tracking	N/A	N/A	Yes	Yes	Yes	*	*	Yes	Yes
Support for 'varying multiplicity' data	N/A	N/A		Yes			N/A		Yes
Uniformity between computed and uncomputed data	Yes	N/A	Yes		Yes				*
Caching/storage of experiment point sets		Yes	Yes	*	Yes			Yes	Yes
Variable priority for data computation	Yes					*	Yes		*
Workflow management									
Experiment specification not restricted to DAG	N/A	N/A		*		Yes	*		Yes
Support for domain=range	N/A	N/A		*		Yes	*		Yes
Adaptivity of workflow	N/A	*	Yes	Yes				Yes	Yes
DBMS as workflow management engine						Yes			Yes
Support for failure management	Yes	Yes		*		Yes	Yes		Yes
Stream processing	N/A	Yes					*		*
Change management/versioning					Yes			Yes	
Parallelism	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Table 6.1: Matrix of features provided by different workflow and experiment management systems.

for a specific version, or different components of certain versions could be deemed incompatible and thus fail if the user attempts to bind them in a complex experiment.

Change management at the workflow execution level captures dynamic changes in the execution graph. Currently, these changes are only partially captured due to the fact that the complex experiments are implicitly linked with the aid of domain and range, and postaction. This reflects the design decision of BREW of focusing on modeling data transformations, and not on the execution chain. The latter can be modeled as proposed in Chapter 4, or it can be extracted from the information stored in the system tables for the data transformation tree; specifically, queries could be written to enumerate the entire history of components and postactions that were applied given a starting experiment point, or that refer to an experiment final result. Such a query would recursively track the data using the `experiment_point_correspondence` and build an execution tree. Changes in the workflow execution graph would then be reflected by merging the trees for the entire experiment point set, and showing the different divergence points.

An obvious side effect of BREW's focus on automation is the lack of user interaction capabilities with the component. While the interaction model is designed so as to mimic the user for an executing component, by binding to the standard input and standard output of the user program, the data used during the communication is statically stored in the database, from the component's point of view. Cases where live user interaction with a component is needed are foreseeable. For example, an optimization can present intermediary results and then wait for a decision of whether to continue down the same path or change computing methods. User interaction could then be modeled either through special domain types or a customized postaction function. Extreme care has to be taken however to ensure a consistent treatment of distributed and parallel executing experiments.

BREW could benefit from a graphical interface to simplify the interaction even more. Due to the reduced amount of work that has to be performed to design and manage complex experiments, a frontend can be created allowing the user to 'drag and drop' components and graphically manage the interactions. A simple graph drawing tool and a way to specify domain and ranges would be sufficient, although making use of more advanced features would require support for the postaction function, views, and other SQL elements. Different views for an experiment's progress can be defined, as well as tools to analyze and annotate the results.

Finally, BREW's data model could also be improved with an eye toward supporting dependencies be-

tween experiment points belonging to the same experiment, better stream processing and better support for computational steering [14]. Currently all these features can be modeled by BREW, but there are no system functions defined to specifically aid any of them. Such features would first require a better formalization of the data model to explicitly differentiate between types of information. Currently BREW is ignorant of the data stored and processed in the system for any other purpose apart from managing component interaction and execution. Adding such features, though, would require BREW to be bound to a specific experiment class, due to the domain knowledge that usually has to be integrated.

Appendix A

Complete Codes for BREW Experiments

This Appendix presents the complete code needed to declare the experiments presented in the previous chapter. The primary language used is PostgreSQL, an extended version of SQL. There are also parts of functions (such as postactions) written in TCL.

A.1 Similarity Search

```
\set start_proc_cmd '\'/home/pgsql/start_process -h
       localhost -U emtry -D emtry -l emtry.log -o\''
\set stop_proc_cmd '\'/home/pgsql/stop_process -h
       localhost -U emtry -D emtry -n\''

-- scenario 1
-- take a document formed out of words and split it
-- search for first 10 documents on google for that word
-- merge all the document lists together and return the most common

-- create the domain
-- because in this case this is the first ep in the series,
-- it uses the system OID as opposed to a separate column for it
CREATE TABLE documents_input (
        content         VARCHAR NOT NULL
) WITH OIDS;

-- create the range
-- since this is a sr/mr experiment (single row input, multiple row output
-- that gets counted as 1 ep
-- we DO NOT have oids given by the system on the range.
CREATE TABLE documents_output (
        name            VARCHAR NOT NULL,
        oid             oid
) WITHOUT OIDS;

-- register the function
```

```
SELECT register_function('documents', 'documents_input', 'documents_output',
                     'function to generate a list of words from a text');

-- register a simple experiment
SELECT register_experiment('docs', 'documents', '',
                  'decompose a document into words for similarity search');


-- second part of the similarity search

-- domain does not need to be created because this uses the
-- range of the previous experiment

-- create the range
-- range does not use the oids of the actual rows, but the oids from the
-- input, or experiment points
-- we only have one field, document retreived for that word
CREATE TABLE words_output (
        retdoc          VARCHAR NOT NULL DEFAULT '',
        oid             oid
) WITHOUT OIDS;

-- register the function
SELECT register_function('words', 'documents_output', 'words_output',
                   'function to generate the list of documents from a word');

-- register a simple experiment
SELECT register_experiment('fetch', 'words', '',
                   'gets the documents for the list of words');


-- third part of the similarity search

-- again domain does not need to be created

-- range for the collector
CREATE TABLE collector (
        best            VARCHAR NOT NULL DEFAULT '',
        oid             oid
) WITHOUT OIDS;

-- register the function
SELECT register_function('collector1', 'words_output',
                        'collector', 'get the best scored document');

-- register a simple experiment
SELECT register_experiment('score', 'collector1','',
                        'end of similarity search');
```

A.2 HPRD Experiment

Following is the HPRD experiment which uses the postaction function to insert names back in its own domain.

```
\set start_proc_cmd '\'/home/pgsql/start_process -h
      localhost -U emtry -D emtry -l emtry.log -o\''
\set stop_proc_cmd '\'/home/pgsql/stop_process -h
      localhost -U emtry -D emtry -n\''

-- create the domain, which contains the protein names
CREATE TABLE hprd_in (
        name            VARCHAR NOT NULL
) WITH OIDS;


-- create the range, which is declared as
-- Multiple Row
CREATE TABLE hprd_out (
        interactor      VARCHAR NOT NULL DEFAULT '',
        oid             oid
) WITHOUT OIDS;


-- the postaction function for the names that have not
-- been previously seen
CREATE OR REPLACE FUNCTION hprd_postaction(VARCHAR(32), INTEGER)
    RETURNS BOOLEAN AS '
    spi_exec "INSERT INTO hprd_in
          SELECT interactor FROM hprd_out AS o, hprd_in AS i,
                experiment_point_correspondence AS epc,
                experiment_points as EP
          WHERE ep.seq_no = '$2' AND ep.id = '$1' AND
                epc.domain_oid = ep.value AND o.oid = epc.range_oid AND
                o.interactor != i.name"
    return 1;
' LANGUAGE 'pltcl';


-- register the function
SELECT register_function('hprd', 'hprd_in', 'hprd_out',
          'function to generate list of interactors for given protein');

-- register a simple experiment
SELECT register_experiment('hprd-test', 'hprd', '',
```

```
          'test HPRD for a couple of proteins');

-- insert several test values
INSERT INTO hprd_in VALUES('Vav');
INSERT INTO hprd_in VALUES('Lat');
INSERT INTO hprd_in VALUES('Cbl');
```

A.3 SCPD Experiments

Following is a description of the SCPD experiments mentioned in the thesis. The first experiment was used to generate the transcriptional regulatory network graph; the rest are variations used to test the performance of the BREW system.

```
\set start_proc_cmd '\'/home/pgsql/start_process -h
        localhost -U emtry -D emtry -l emtry.log -o\''
\set stop_proc_cmd '\'/home/pgsql/stop_process -h
        localhost -U emtry -D emtry -n\''

-- first part of the SCPD sample experiment
-- input table: URL to browse to in order to get the list of factors
-- table is defined with OIDS because it is the first experiment
-- in the series
CREATE TABLE urls (
        url             VARCHAR NOT NULL
) WITH OIDS;

-- output table for the gene getter
-- contains the genes obtained from browsing to the source URL
CREATE TABLE factors (
        name            VARCHAR NOT NULL,
        oid             oid
) WITHOUT OIDS;

-- register the component
-- this is a JAVA program defined in NetScrape, that browses to the URL
-- given on standard in, and prints a list of factors on standard out
SELECT register_function('get_factors', 'urls', 'factors',
       'component to get the factor names from a www site');

-- register a simple experiment on the 'factors' component
-- the experiment has no postaction function because the linking is
-- just SR - SMR
SELECT register_experiment('scpd_factors', 'get_factors','',
                    'get the factors from the SCPD site');

-- insert the URL to be processed in the system
```

```
-- this does not start the experiment, as for demonstration purposes the
-- automatic_executor function has been disabled
INSERT INTO urls VALUES ('http://cgsigma.cshl.org/cgi-bin/jz/getfactorlist');

-- second part of the SCPD experiment
-- getting the regulators for the genes
-- input table: Output table from the above experiment, which
-- contains genes

-- output table
-- stores the (factor name, regulator) tuples
-- table does no longer have system-defined OIDS as the information
-- must be grouped on the name
-- this creates a SR - MR experiment
CREATE TABLE regulated (
        factor_name     VARCHAR NOT NULL,
        element         VARCHAR NOT NULL,
        oid             oid
) WITHOUT OIDS;

-- register the specific component
-- this is another java program written in NetScrape to fetch the list of
-- regulated genes given a specific factor, specific to the SCPD site
SELECT register_function('get_regulated', 'factors', 'regulated',
          'component to return the regulated genes');

-- register a simple experiment
-- we do not need a postaction function
SELECT register_experiment('scpd_regulators', 'get_regulators','',
          'get the regulators for the genes in the SCPD experiment');


-- third part of the experiment
-- creating graphs from the connected components
-- input table : Output table from the previous part, the list of tuples
-- containing the genes and their regulators

-- output table
-- name and file description for the connected components that need to be
-- plotted
-- table has System OIDS again since we want to keep the connected components
-- separate
-- there is no longer a clear connection between the graphs and individual genes
-- if more URLs are used, the graphs can be distinguished using the
-- experiment_point_correspondence
CREATE TABLE connected_components (
        component_name          VARCHAR NOT NULL,
        file                    VARCHAR NOT NULL
```

```
) WITH OIDS;


-- register the specific component
-- this is a C++ program that takes as input the list of edges and
-- prints out connected components as graphs
SELECT register_function('conn_components', 'regulators',
         'connected_components', 'component to split and create graphs');

-- register a simple experiment
-- again we do not need the postaction function
-- if the experimenter required that the graphs be combined in a certain
-- way, the postaction function would be inevitable
SELECT register_experiment('generate_graphs', 'components', '',
            'generate the graphs');

-- last part of the experiment
-- plot the graphs using a visualization tool
-- input table: Output table from the connected components
-- containing the graph definitions

-- output table
-- name and path to the file local to the installation in which the
-- image is found.
-- Table does not have System OIDS, to be associated with the specific
-- graph definition
-- it was derrived from
CREATE TABLE graphs (
        filename        VARCHAR NOT NULL,
        oid             oid
) WITHOUT OIDS;

-- register the component
-- this is a bash script that wrapps around Graphviz tools such as dot
-- and neato to generate the images
SELECT register_function('plot', 'connected_components', 'graphs',
          'component to generate the plot for a given description');

-- register a simple experiment
SELECT register_experiment('plot_regulators', 'plot', '',
          'plot the regulatory links between elements in SCPD');
```

A.3.1 Short experiment to test Multiple Row SCPD

This variant of the basic experiment executes each component once on the input list.

```
\set start_proc_cmd '\'/home/pgsql/start_process -h
        localhost -U emtry -D emtry -l emtry.log -o\''
\set stop_proc_cmd '\'/home/pgsql/stop_process -h
        localhost -U emtry -D emtry -n\''

CREATE TABLE scpd_input (
        content         VARCHAR NOT NULL
) WITH OIDS;

-- create the range
CREATE TABLE scpd_output (
        name            VARCHAR NOT NULL,
        oid             oid
) WITHOUT OIDS;

-- register the function
SELECT register_function('scpd', 'scpd_input', 'scpd_output', 'function');

-- register a simple experiment
SELECT register_experiment('fetch', 'scpd','', 'decompose');

INSERT INTO scpd_input VALUES ('try');

CREATE TABLE reg_output (
        gn              VARCHAR NOT NULL,
        ret             VARCHAR NOT NULL,
        oid             oid
) WITHOUT OIDS;

-- register the function
SELECT register_function('reg', 'scpd_output', 'reg_output', 'function');

-- register a simple experiment
SELECT register_experiment('get', 'reg','', 'get');
```

A.3.2 SCPD experiment to test the overhead of starting a component

```
\set start_proc_cmd '\'/home/pgsql/start_process -h
        localhost -U emtry -D emtry -l emtry.log -o\''
\set stop_proc_cmd '\'/home/pgsql/stop_process -h
        localhost -U emtry -D emtry -n\''

CREATE TABLE scpd_input (
        content         VARCHAR NOT NULL
) WITH OIDS;
```

```
-- create the range
CREATE TABLE scpd_output (
        name            VARCHAR NOT NULL
) WITH OIDS;

-- register the function
SELECT register_function('scpd', 'scpd_input', 'scpd_output', 'function');

-- register a simple experiment
SELECT register_experiment('fetch', 'scpd','', 'decompose');

INSERT INTO scpd_input VALUES ('try');

CREATE TABLE reg_output (
        gn              VARCHAR NOT NULL,
        ret             VARCHAR NOT NULL,
        oid             oid
) WITHOUT OIDS;

-- register the function
SELECT register_function('reg', 'scpd_output', 'reg_output', 'function');

-- register a simple experiment
SELECT register_experiment('get', 'reg','', 'get');

CREATE OR REPLACE FUNCTION start() returns integer as '
        spi_exec "select start_experiment_executor(''fetch'')"
        spi_exec "select start_experiment_executor(''get'')"
        return 1
' language 'pltcl';
```

A.3.3 SCPD experiment to try different parallel configurations

```
\set start_proc_cmd '\'/home/pgsql/start_process -h
        localhost -U emtry -D emtry -l emtry.log -o\''
\set stop_proc_cmd '\'/home/pgsql/stop_process -h
        localhost -U emtry -D emtry -n\''

CREATE TABLE scpd_input (
        content         VARCHAR NOT NULL
) WITH OIDS;

-- create the range
CREATE TABLE scpd_output (
        name            VARCHAR NOT NULL
) WITH OIDS;
```

```
-- register the function
SELECT register_function('scpd', 'scpd_input', 'scpd_output', 'function');

-- create postaction
CREATE OR REPLACE FUNCTION copyover(VARCHAR(32), INTEGER)
    RETURNS BOOLEAN AS '
    set np [spi_exec "INSERT INTO reg_input
          SELECT name FROM scpd_output AS o, scpd_input as i,
                experiment_point_correspondence AS epc,
                experiment_points as ep
          WHERE ep.seq_no = ''$2'' AND ep.id = ''$1'' AND
                epc.domain_oid = ep.value AND o.oid = epc.range_oid"]
    incr np
    for {set x 0} {$x<$np} {incr x} {
      spi_exec "SELECT automatic_executor(''get'')"
    }
    return 1;
' LANGUAGE 'pltcl';


-- register a simple experiment
SELECT register_experiment('fetch', 'scpd','copyover', 'decompose');

INSERT INTO scpd_input VALUES ('try');

CREATE TABLE reg_input (
        name            VARCHAR NOT NULL
) WITH OIDS;

CREATE TABLE reg_output (
        gn              VARCHAR NOT NULL,
        ret             VARCHAR NOT NULL,
        oid             oid
) WITHOUT OIDS;

-- register the function
SELECT register_function('reg', 'reg_input', 'reg_output', 'function');

-- register a simple experiment
SELECT register_experiment('get', 'reg','', 'get');

UPDATE experiments SET np = 6 WHERE id = 'get';

create or replace function start() returns integer as '
        spi_exec "select start_experiment_executor(''fetch'')"
        return 1
' language 'pltcl';
```

A.4 Promoter Analysis

Following is a description of the promoter analysis experiment mentioned in the thesis.

```
-- example of a Promoter Analysis complex experiment formed of
-- two components working on the hs_chr21 and
-- hs_chr22 chromosomes respectively

-- create the domain
-- notice that unlike other experiments, even if this is the first
-- domain in the link of complex experiments, it does not have
-- system assigned OID values. This behavior is supported by the
-- special trigger function derrived from the default one
CREATE TABLE motifs_table (
        name            VARCHAR NOT NULL,
        sequence        VARCHAR NOT NULL,
        minimum         VARCHAR NOT NULL,
        maximum         VARCHAR NOT NULL,
        oid             oid
) WITHOUT OIDS;

-- create the range
-- range has OIDS since clustering these entries based on what motif
-- they originated from is not necessary. All information needed to
-- reason the results is included as various output columns
CREATE TABLE search_output (
        locus_name              VARCHAR NOT NULL,
        gene_name               VARCHAR NOT NULL,
        element_strand          VARCHAR NOT NULL,
        gene_strand             VARCHAR NOT NULL,
        element_name            VARCHAR NOT NULL,
        element_sequence        VARCHAR NOT NULL,
        abs_seq_pos             VARCHAR NOT NULL,
        upstream_pos            VARCHAR NOT NULL,
        annotation              VARCHAR NOT NULL,
        protein                 VARCHAR NOT NULL
) WITH OIDS;

-- register the function
SELECT register_function('upstream-hs_chr21', 'motifs_table',
            'search_output', 'function to look in hs_chr21');

-- register a simple experiment
SELECT register_experiment('hs_chr21', 'upstream-hs_chr21','',
```

```
            'test for the component');

-- add another installation, this time remote
INSERT INTO installations
    (id, description, data_dir, sh, clean_trash_cmd)
VALUES (
    'bioinformatics',
    'remote machine for running experiments',
    '~/upstream',
    '/usr/bin/ssh -l brew bioinformatics.cs.vt.edu ',
    'clean_trash -h bioinformatics.cs.vt.edu -U emtry -D emtry -o'
);

-- register the function
SELECT register_function('upstream-hs_chr22', 'motifs_table',
    'search_output', 'function to look in hs_chr22');

-- change the original installation
DELETE FROM component_installations WHERE component = 'upstream-hs_chr22';
INSERT INTO component_installations
    (component, installation, start_exec_cmd, start_exec_arg, stop_exec_cmd)
VALUES
    ('upstream-hs_chr22', 'remote on bioinformatics',
     '$HOME/upstream/start_process -h www.croconet.net -U
          emtry -D emtry -l emtry.log -o
     'upstream-hs_chr22',
     '$HOME/upstream/stop_process -h www.croconet.net -U emtry -D emtry -n');

-- register a simple experiment
SELECT register_experiment('hs_chr22', 'upstream-hs_chr22', '',
    'test for the component');

-- modify the default_experiment_point trigger function
CREATE OR REPLACE FUNCTION multiple_experiment_point()
    RETURNS OPAQUE AS '
        DECLARE
            seq integer;
        BEGIN
            SELECT nextval(''default_point_seq_no'') INTO seq;

            INSERT INTO experiment_point_states(id, seq_no, state)
                VALUES (TG_ARGV[0], seq, 0);
            INSERT INTO experiment_points(id, seq_no, value)
                VALUES (TG_ARGV[0], seq, NEW.OID);

            INSERT INTO experiment_point_states(id, seq_no, state)
                VALUES (TG_ARGV[1], seq, 0);
            INSERT INTO experiment_points(id, seq_no, value)
```

```
                VALUES (TG_ARGV[1], seq, NEW.OID);
            return NEW;
        END
' language 'plpgsql';

-- and update the trigger
DROP TRIGGER default_experiment_trigger_ ON motifs_table;
CREATE TRIGGER multiple_experiment_trigger
    AFTER INSERT ON motifs_table FOR EACH ROW
    EXECUTE PROCEDURE default_experiment_point('hs_chr21', 'hs_chr22');

INSERT INTO motifs_table VALUES ('TonE', 'TGGAAAAWHWH', '1');
```

Appendix B

BREW Support Codes and Schemas

Following is a selection of the system code supporting the functionality presented for BREW in the previous chapters. Portions of this code are written in Tcl, both embedded as the PlTcl language supported by the PostgreSQL database management system, and as standalone code. A significant part of the code is written in SQL and its extended variant implemented by PostgreSQL.

```
-- register a function with the system
--  component - name of the function
--  domain - table or view that defines and stores the function input
--  range - table that defines and stores the function output
--  description - information about the function
CREATE OR REPLACE FUNCTION register_function
        (varchar(32), varchar, varchar, text)
        RETURNS integer AS '
    spi_exec "INSERT INTO components
        (id, description, domain, range)
              VALUES
        (''[quote $1]'', ''[quote $4]'', ''[quote $2]'', ''[quote $3]'')"
    spi_exec "INSERT INTO component_installations
        (component, installation, start_exec_cmd, start_exec_arg,
        stop_exec_cmd)
            VALUES
        (''[quote $1]'', ''default'',
        ''/home/pgsql/start_process -h localhost -U emtry -D emtry \
          -l emtry.log -o'', ''[quote $1]'', ''/home/pgsql/stop_process -h \
          localhost -U emtry -D emtry -n'')"
    return 1;
' LANGUAGE 'pltcl';
```

```
-- general structure for the experiment points
-- in addition to the references to experiment (id) and the actual number
-- of the point (seq_no) this stores the reference to the input element,
-- using the OID.
-- the referred experiment point can be single-row or multi-row
```

```
CREATE TABLE experiment_points (
    id                varchar(32)   NOT NULL CHECK (id > ''),
    seq_no            integer       NOT NULL DEFAULT '0',
    value             oid           NOT NULL,
    PRIMARY KEY (id, seq_no),
    FOREIGN KEY (id, seq_no)
        REFERENCES experiment_point_states (id, seq_no)
        ON DELETE CASCADE
);

CREATE SEQUENCE default_point_seq_no start 1;

-- create the necessary structures to support a new experiment point
-- in a controller situation, this would be taken care of by the
-- "fresh_experiment_point"
-- outside the EM structure (since this is done for the first experiment
-- in the loop/series) we cannot use that function, so we have a trigger
CREATE OR REPLACE FUNCTION default_experiment_point()
RETURNS OPAQUE AS '
DECLARE
seq integer;
BEGIN
SELECT nextval(''default_point_seq_no'') INTO seq;
INSERT INTO experiment_point_states(id, seq_no, state)
VALUES (TG_ARGV[0], seq, 0);
INSERT INTO experiment_points(id, seq_no, value)
VALUES (TG_ARGV[0], seq, NEW.OID);
return NEW;
END
' language 'plpgsql';


-- function to run automatic executors, up to the number specified in np
-- when a new point is added into the experiment
-- integrated into a rule as on insert on experiments
--  id - name of experiment to create rule on
CREATE OR REPLACE FUNCTION automatic_executor(varchar(32))
        RETURNS BOOLEAN AS '
        spi_exec "SELECT count(id) AS exist FROM experiment_executors
                 WHERE exp_id = ''[quote $1]''"
spi_exec "SELECT np FROM experiments WHERE id = ''[quote $1]''"
        if {$exist < $np} {
                # elog NOTICE "Starting executor for [quote $1]";
                spi_exec "SELECT start_experiment_executor(''[quote $1]'')"
        }
        return true
' language 'pltcl';
```

```
-- register a new experiment. components must be instantiated as
-- experiments before any computation can be done with them
--  id - name of the experiment, must be simple [a-zA-Z0-9_] for now
--       because of the trigger creation
--  component   - function used by this experiment, must be already
--       defined in components
--  postaction  - function to call after the experiment point has been
--       computed
--  description - anything that the user wants to insert as description
--       for this experiment
-- the function also registers a trigger to create the default state for
-- the experiment point.
CREATE OR REPLACE FUNCTION register_experiment(varchar(32), varchar(32),
        varchar(32), text)
    RETURNS integer AS '
    # register the experiment as given
    spi_exec "INSERT INTO
        experiments (id, component, postaction, description)
            VALUES
        (''[quote $1]'', ''[quote $2]'', ''[quote $3]'', ''[quote $4]'')"

    # see if we have an experiment before us, and if not, create the
    # default_experiment_point
    # so the structure is created automatically
    spi_exec "SELECT domain AS domain FROM components WHERE
              id = ''[quote $2]''"
    if {[spi_exec "SELECT experiments.id
            FROM components as a, components as b, experiments
            WHERE a.domain=''[quote $domain]'' AND
                  a.domain = b.range AND
                  experiments.component=b.id"] != 1} {

        spi_exec "CREATE TRIGGER [join default_experiment_trigger_ $1]
                  AFTER INSERT ON $domain FOR EACH ROW
                  EXECUTE PROCEDURE
                      default_experiment_point(''[quote $1]'')"
    }
    # create the rule for automatic_executor;
    spi_exec "CREATE RULE executor_trigger AS ON INSERT TO $domain DO
              SELECT automatic_executor(''[quote $1]'')"
    return 1
' LANGUAGE 'pltcl';
--


-- create the correspondence table for multirow experiment points and
-- experiments. this table is used when the source experiment point is
-- computed to a series of destination experiment points, that by the
```

```
-- nature of the application need to be recorded as separate experiment
-- points. Therefore we have to register them under new OIDs, that we
-- have to keep track of as part of data management to make usage of this
-- feature one must simply make sure that the range table is
-- declared to be WITH OIDS;
CREATE TABLE experiment_point_correspondence (
domain_oid OID,
range_oid OID
);

-- end of file


-- template for the executor/controller from each every component is
-- executed

#!/bin/sh
# execute this script with a Tcl interpreter \
exec expect $0 -- ${1+"$@"}

# variables:
# script - name of the file/script/executor to be displayed for
# diagnostic purposes
# the transaction at domain reading might last too long for very large
# multirow exps

# load packages
package require emdag
namespace import emdag::*


# print usage info and exit
proc usage {} {
    puts stderr "usage: %script [connusage] executor_id"
    exit 1
}


# parse command line
catch {
    if {[string equal [lindex $argv 0] --]} {set argv [lrange $argv 1 end]}
}
if {[catch { set conninfo [conninfo argv connargv] } msg]} {
    puts stderr $msg
    usage
}
set i 0
foreach {arg val} $argv {
```

```
    switch -glob -- $arg {
        -- {
            incr i
            break
        }
        -* {
            puts stderr "%script: invalid argument: $arg"
            usage
        }
        default {
            break
        }
    }
    incr i 2
}
set argv [lrange $argv $i end]
if {[llength $argv] != 1} usage
set executor_id [lindex $argv 0]


# connect to the database and register the executor
if {[catch { set db [pg_connect -conninfo $conninfo] } msg]} {
    puts stderr "%script: pg_connect: $msg"
    exit 1
}
register_experiment_executor $db $executor_id \
    experiment_id component installation debug

puts stderr "doing something"

# compute points
while {1} {

    # grab a point to compute
    set seq_no [grab_experiment_point $db $executor_id]
    if {$seq_no < 0} { break }
    if {$debug > 0} {
        experiment_point_diagnostic $db $experiment_id $seq_no debug \
            "grabbed by %script $executor_id"
    }

    # read what point we are working on
    pg_exec_check $db "BEGIN"
    pg_select $db \
        "SELECT value
         FROM experiment_points
         WHERE id = '[quote_sql $experiment_id]' AND seq_no = $seq_no" a {
        set oid $a(value)
```

```
}
# first get the domain and range names
pg_select $db \
    "SELECT cp.domain, cp.range
     FROM experiment_points ep, experiments e, components cp
     WHERE ep.seq_no = $seq_no and ep.id = e.id and cp.id = e.component"
    a {
        set domain $a(domain)
        set range $a(range)
}
# and figure out if we have OIDs for the range, in case this is a
# multirow experiment
pg_select $db \
    "SELECT relhasoids
     FROM pg_class
     WHERE relname='[quote_sql $range]'" a {
    set range_oids $a(relhasoids)
}

# see if we have to be a controller with this
# the condition is for our output to appear as the input to another
# function
pg_select $db \
    "SELECT experiments.id
     FROM components as a, components as b, experiments
     WHERE a.range='[quote_sql $range]' AND
           a.range = b.domain AND
           experiments.component=b.id" a {
    set next_experiment $a(id)
}

if {[info exists next_experiment]} {
    # then we have something follow this, define and register a
    # controller. it gets registered on the next_experiment
    set result [pg_exec $db \
        "INSERT INTO experiment_controllers (exec_id, exp_id)
        VALUES ($executor_id, '[quote_sql $next_experiment]')"]

    set msg [pg_result $result -error]
    if {![string equal $msg ""]} {
        catch {pg_exec_check $db "ROLLBACK"}
        experiment_point_diagnostic $db $experiment_id $seq_no \
            error "cannot register controller: $executor_id, $msg"
        release_experiment_point $db $executor_id crashed
        continue
    }
    pg_select $db "SELECT id AS controller_id FROM
                        experiment_controllers
```

```
                    WHERE oid = '[quote_sql [pg_result $result -oid]]'"
        a {
         set controller_id $a(controller_id)
    }
    pg_exec_check $db \
        "SELECT register_experiment_controller($controller_id)"
    pg_result $result -clear
    # done registering the controller

    # set the control variables
    pg_select $db "SELECT
        experiment_controller_stop_cond('[quote_sql $experiment_id]')
            AS c1,
        experiment_point_done_cond('[quote_sql $next_experiment]')
            AS c2" a {
        set c1 $a(c1)
        set c2 $a(c2)
    }

    # register these so we know when WE have been shut down, or when
    # slave points computed
    pg_listen $db $c1 { set emdag::emdag_event 1 }
    pg_listen $db $c2 { set emdag::emdag_event 1 }

    if {$debug > 0} {
        experiment_point_diagnostic $db $experiment_id $seq_no debug \
            "controller $controller_id started"
    }
}
# done with the part specific to the controller

set crashed 0

# execute the script
set stty_init -echo
set timeout -1
log_user 0
if {[catch {
    if {$debug > 0} {
        experiment_point_diagnostic $db $experiment_id $seq_no debug \
            "spawning %script_code"
    }
    eval spawn %script_code
} msg]} {
    set crashed 1
    experiment_point_diagnostic $db $experiment_id $seq_no error $msg
    pg_exec_check $db "COMMIT"
} else {
```

```
puts stderr "%script executor working on $oid"
# set up the script by sending all the input columns in the order
# we got them
# excluding the oid column
# the input part is here to take care of multi-row input points
# we don't look at the input oid, it only matters for output
# the end of column and and of row can be modified to be only
# "end of line" and "blank line" for most applications.
# only when the target app. requires sending free text with
# end-of-line in it do the <--END OF --> structures actually make
# sense
pg_select $db \
    "SELECT *
    FROM $domain
    WHERE oid = $oid" a {
    foreach colname $a(.headers) {
        if {$colname != "oid"} {
            send -- "$a($colname)\r<--END OF COLUMN-->\r"
        }
    }
}
pg_exec_check $db "COMMIT"
# send sort of an "end" marker to help certain
# things if more complicated I/O is needed, then consider sending
# some DB identifier
# and doing the fetching/inserting externally
send -- "\r<--END OF ROW-->\r"

# build up a regular expression for output, based on how many
# output columns we have. since there is no more overlap of the
# domain and the range (since we have to give up the view model
# in order to support multi-row input/output) we do not have to do
# set operations on the input and output columns
# however, we do have to take care of the OID column, so filter
# that out
set range_columns {}
set opt_re {}
pg_select $db \
    "SELECT a.attname AS field
     FROM pg_class c, pg_attribute a, pg_type t
     WHERE c.relname::varchar = '$range' AND
           a.attnum > 0 AND a.attrelid = c.oid AND
           a.atttypid = t.oid
     ORDER BY a.attnum" a {
    if {$a(field) != "oid"} {
        lappend range_columns $a(field)
        append opt_re {([^\r\n]*)\r?\n}
```

```
    }
}

# as long as we have more output coming out, we treat it as
# multi-row output for the same experiment point, and insert
# everything
while {1} {
  expect {
    -re $opt_re {
      set match_list [eval "regexp -inline -- {$opt_re} \
          {$expect_out(0,string)}"]
      set rindex 1
      # check and if the relation did have OIDs then we must
      # let the system give the value and simply update the
      # mapping table experiment_point_correspondence
      if {$range_oids == "f"} {
       set update_query "INSERT INTO $range (oid"
      } else {
          set update_query "INSERT INTO $range ("
      }
      set first 1
      foreach outcol $range_columns {
          if {$first == 1 && $range_oids == "t"} {
              append update_query "$outcol"
              set first 0
          } else {
              append update_query ", $outcol"
          }
      }
      if {$range_oids == "f"} {
          append update_query ") VALUES ($oid"
      } else {
          append update_query ") VALUES ("
      }
      foreach outcol $range_columns {
          if {$range_oids == "t" && $rindex == 1} {
              append update_query "'[quote_sql [lindex \
                  $match_list $rindex]]'"
          } else {
              append update_query ", '[quote_sql [lindex \
                  $match_list $rindex]]'"
          }
          incr rindex 1
      }
      append update_query ");"

      # if we do have the columns thing here we must insert
      # the OID in the mapping table
```

```
# we cannot use the pg_exec_check here because it
# clears the result, which we do need
set result [pg_exec $db $update_query]
set msg [pg_result $result -error]
set new_oid [pg_result $result -oid]
pg_result $result -clear
if {![string equal $msg ""]} {
    error $msg
}
if {$range_oids == "t"} {
      pg_exec_check $db \
        "INSERT INTO experiment_point_correspondence
        (range_oid, domain_oid)
        VALUES
        ($new_oid, $oid);"
}
# see if we are a controller, and if so do the required
# logic for that
# we only get in this case if we had "single multi row"
if {[info exists next_experiment] && $range_oids == "t"} {
  pg_exec_check $db "BEGIN"

  # make a new ep in the next experiment
  pg_select $db \
    "SELECT fresh_experiment_point($controller_id) AS
     iter_no" a {
    set iter_no $a(iter_no)
  }

  # insert the point with the given values
  pg_exec_check $db \
      "INSERT INTO experiment_points
         (id, seq_no, value)
      VALUES
        ('$next_experiment', '$iter_no', '$new_oid');"
  pg_exec_check $db "COMMIT"
  # is already an executor running on the second experiment
  pg_exec_check "SELECT automatic_executor('$next_experiment');"

  # wait for experiment point outputs
  while {1} {
    catch {unset state}
    # see if point got calculated
    pg_select $db \
      "SELECT state FROM experiment_point_states
       WHERE id = '$next_experiment' AND
       seq_no = $iter_no" a {
           set state $a(state)
```

```
                }
                if {[info exists state]} {
                    if {$state == 2} {
                       # point computed
                       break
                    } elseif {$state == 3} {
                        # point crashed
                        set crashed 1
                        experiment_point_diagnostic $db $experiment_id \
                          $seq_no error "slave point $iter_no crashed"
                          release_experiment_point $db $executor_id crashed
                          break
                    } else {
                        # still computing...
                        catch {unset emdag::emdag_event}
                    }
                } else {
                    # point cannot be found, was clearly deleted
                    set crashed 1
                    experiment_point_diagnostic $db $experiment_id $seq_no \
                      error "slave point $iter_no was apparently deleted"
                    release_experiment_point $db $executor_id crashed
                    break
                }
            }
        }
     }
     eof {
         break
     }
  }
}
# done with the output

# see if we had no range oids, in which case we have to do the controller
# logic here when we are done with the ep
# because in the muti-row case
if {[info exists next_experiment] && $range_oids == "f"} {
    pg_exec_check $db "BEGIN"

# make a new experiment point in the next experiment
    pg_select $db \
        "SELECT fresh_experiment_point($controller_id) AS iter_no" a {
        set iter_no $a(iter_no)
    }

    # insert the point with the given values
    # here we use the old oid
```

```
    pg_exec_check $db \
          "INSERT INTO experiment_points
                  (id, seq_no, value)
           VALUES
                  ('$next_experiment', '$iter_no', '$oid');"
      pg_exec_check $db "COMMIT"
      pg_exec "SELECT automatic_executor('$next_experiment');"

      # wait for experiment point outputs

 -- part removed, for a complete listing check the code distribution

   }

   catch {close}
}
 # store results for the recently computed point if ok
if {!$crashed} {
   if {$debug > 0} {
        experiment_point_diagnostic $db $experiment_id $seq_no \
       debug "computed by %script $executor_id"
    }
   release_experiment_point $db $executor_id computed
   # call the postaction
   pg_select $db \
        "SELECT postaction FROM experiments
        WHERE id='[quote_sql $experiment_id]'" a {
        set postaction $a(postaction)
   }
    if {$postaction != ""} {
       # we have a postaction, go ahead and call it with required params
        pg_select $db \
           "SELECT $postaction('[quote_sql $experiment_id]', $seq_no)
             AS result" a {
           set postaction_result $a(result)
        }
        experiment_point_diagnostic $db $experiment_id $seq_no \
        debug "postaction returned with a value of $postaction_result"
   }
}

# clean up results if it crashed
if {$crashed} {
  if {$debug > 0} {
       experiment_point_diagnostic $db $experiment_id $seq_no \
        debug "crashed - %script executor_id"
  }
    release_experiment_point $db $executor_id crashed
```

```
}

}


# clean up and exit
unregister_experiment_executor $db $executor_id
catch { pg_close $db }
exit 0


# end of file

-- part removed, for a complete listing please see the code distribution
```

Appendix C

NetScrape: Batch Browsing for Experiment Management

NetScrape is a Java library for simulating a web browser session. It allows a programmer to rapidly generate 'web scrapers' designed to query an online database or interactive service for specific information. The programmer is freed from having to consider many elements of the browser-server interaction such as cookies, redirection, referring urls, user agents, and form submission. Some of the commands used for browsing and interaction with the system are presented in Table C.1.

This appendix contains the code for the component programs designed in NetScrape. The code is written entirely in JAVA, but some basic HTML terms and constructs are needed for a complete understanding. For instance, below is part of the output of the `printTree` command on the semantic tree generated upon browsing the main page on the 'www.vt.edu' site:

```
Prospective Students
Current Students
Faculty &amp;amp; Staff
Alumni & Friends
Family &amp;amp; Visitors
Business &amp;amp; Industry

Extended Campus
---Hampton Roads
---National Capital Region
---Richmond
---Roanoke
---Southwest Virginia
------Friday, April 15, 2004
---------Blacksburg
------: Mostly Cloudy, 66&amp;deg;F (18&amp;deg;C)
------News
---------Advisory Council for Campus Environmental Sustainability formed
---------Cadets donate $29,000 to D-Day Memorial
---------more...
------Events
---------Lecture on Soul, Salsa, and South Atlantic Migrations
---------Calendar
```

Function	Description
class State	Models a web browser that can accept different user commands
void visitURL(String)	Changes the session state to a particular url
void clickTextLink(String)	Simulates a click on a text or image link
void setMetaRefresh(boolean)	Determines whether or not NetScrape follows meta redirection
chooseForm(String/int)	Selects a form from a document using an index or naming identifier
void submitForm()	Submits the selected form (or the first form if no selected form)
void choose<Control>(String/int)	Selects a particular form element control from within the current form
void <act><Control>(...)	Performs some action on a particular form element control. Available actions depend on the control type (button, radio, text field, etc)
String getHTML()	Returns the HTML of the current page in the state
String getURL()	Returns the URL of the current page in the state
class Parser	Builds a table with matches to particular templates for specific information
class semanticTree(String)	Builds the semantic tree of a HTML page.
String printTree()	Given an semanticTree initialized with an HTML document, attempts to extract a tree of relevant text relations in the HTML page based on designer's tagging choices and prints out groups of related text objects in tree form

Table C.1: NetScrape common classes and commands.

```
------Featured Sites &amp;amp; Initiatives
---------Campus Update video
---------Commencement Information
------Awards
---------Goldwater scholars have high goals as bio-researchers
---------Researcher of the Week
------A to Z Index
------WebCam
------Campus Map
------Choose a Quicklink:
------4Help
------Blackboard
------Blacksburg Electronic Village
------Campus Map
...
```

Throughout the execution of various commands, the system builds up an internal tree structure of each document's interactive elements. It also maintains other state information about the entire session such as previously visited URLs, stored and changed cookies, and submission values.

Some of NetScrape's limitations are due to an interface restricted to standard human-browser-server interactions. Interaction systems not supported include Flash plugins, JAVA elements, embedded Javascript, and ActiveX controls. Most of these controls however are not frequently present on sites that are targeted for automation, such as on-line databases, with the exception of Javascript. At its current implementation NetScrape is quite limited by this lacking support, but more functionality might be added in future versions. Furthermore, the current version's use of a non-threaded application model results in NetScrape programs to hang indefinitely upon reaching a non-responding site.

C.1 SCPD Components

```
// component program to get the list of transcriptional factors
// from the given site
// only uses the "browse to link" feature, and extracting
// information from the HTML file
import java.io.*;
import java.util.*;
import wrapper.*;

public class scpd {
    public static void main(String[] args) throws Exception {

        String addr= new String();
        BufferedReader stdin =
            new BufferedReader(new InputStreamReader(System.in));
        try { addr = stdin.readLine();}
        catch (Exception e) { e.printStackTrace();}

        //perform navigation to output file
        State s=new State(addr);
```

```
        s.setMetaRefresh(false);

        //set up a template pattern to get values from the HTML output
        String pat="/cgi-bin/jz/getfactor[?]%%factor%%\"";

        //create a parser with the pattern
        Parser p=new Parser(pat);
        //get those values in a vector of name/value pairs
        Vector v=p.runDocument(s.getHTML());
        //use the Parser to extract the fields
        ArrayList cont = p.getDataset(v);
        for (int i = 0; i < cont.size(); i++)
            System.out.println(
                (String)((HashMap)cont.get(i)).get("factor"));
    }
}


// component program to get the list of regulated genes given
// a certain transcription factor
// this component presents advanced functionality such as
// working with forms, clicking on links and buttons
import java.io.*;
import java.util.*;
import wrapper.*;

public class scpd_elements {
    public static void main(String[] args) throws Exception {

        String addr= new String();
        BufferedReader stdin =
            new BufferedReader(new InputStreamReader(System.in));
        while (true) {
            try { addr = stdin.readLine();}
            catch (Exception e) { e.printStackTrace();}

            if (addr.compareTo("") == 0) break;

            //perform navigation to output file
            State s=new State("http://cgsigma.cshl.org/cgi-bin/jz/getfactor?"+
                    addr);
            s.chooseForm(0);
            s.chooseButton("Get regulated genes");
            s.submitForm();
            String pat="/cgi-bin/jz/getgene2[?]%%factor%%\"";

            //create a parser with the pattern
            Parser p=new Parser(pat);
```

```
            //get those values in a vector of name/value pairs
            Vector v=p.runDocument(s.getHTML());
            //use the Parser
            ArrayList cont = p.getDataset(v);
            for (int i = 0; i < cont.size(); i++)
                System.out.println(addr + "\n" +
                    (String)((HashMap)cont.get(i)).get("factor"));
        }
    }
}
```

C.2 Similarity Search

```
// component program to run a search on the www.google.com website, and
// return the links of the documents on the first page of results
// uses form picking and filling for it's functionality
import java.io.*;
import java.util.*;
import wrapper.*;

public class google {
    public static void main(String[] args) throws Exception {

        BufferedReader stdin =
            new BufferedReader(new InputStreamReader(System.in));
        while (true) {
            String addr= new String();
            try { addr = stdin.readLine();}
            catch (Exception e) { e.printStackTrace();}

            if (addr == null || addr.compareTo("") == 0) break;

            // perform navigation to output file
            State s=new State("http://www.google.com");
            s.chooseForm(0);
            s.chooseTextBox(0);
            s.changeText(addr);
            s.submitForm();

            String pat=
             "<p class=g><a href=%%url%% onmousedown=\"return clk";

            // create a parser with the pattern
            Parser p=new Parser(pat);
            // get those values in a vector of name/value pairs
```

```
            Vector v=p.runDocument(s.getHTML());
            // use the Parser output function to print
            // formatted to standard out
            p.printDataset2(v);
        }
    }
}
```

C.3 HPRD Component

```
// component to get the list of interactions for a protein on the
// HPRD site
import java.io.*;
import java.util.*;
import wrapper.*;

public class hprd {
    public static void main(String[] args) throws Exception {

        String addr= new String();
        BufferedReader stdin =
            new BufferedReader(new InputStreamReader(System.in));
        try { addr = stdin.readLine();}
            catch (Exception e) { e.printStackTrace();}

        //perform navigation
        State s=new State("http://www.hprd.org/query");
        s.chooseForm(0);
        s.chooseTextBox(0);
        s.changeText(addr);
        s.submitForm();
        s.clickTextLink("INTERACTIONS");
        // System.out.println(s.getHTML());

        String pat="<td bgcolor=\"#f9f9f9\" height=\"20\"" +
                   " nowrap>\\s*<span>\\s*<span>\\s*<a " +
                   "href=\"/protein/\\d*\\?selectedtab=INTERACTIONS\">" +
                   "%%name%%</a>\\s*</span>";

        //create a parser with the pattern
        Parser p=new Parser(pat);
        //get those values in a vector of name/value pairs
        Vector v=p.runDocument(s.getHTML());
        //use the Parser output function to print formatted to standard out
        p.printDataset2(v);
    }
}
```

C.4 Promoter Analysis

The case study involving promoter analysis does not contain a NetScrape component, as there is no part of the application that requires data to be extracted from webpages.

Bibliography

[1] A. Ailamaki, Y. Ioannidis, and M. Livny. Scientific Workflow Management by Database Management. In *Proceedings of the 10th International Conference on Scientific and Statistical Database Management*, pages 190–199. IEEE Computer Society, 1998.

[2] P. Avery and I. Foster. Petascale Virtual-Data Grids for Data Intensive Science. White Paper, http://www.griphyn.org, 2000.

[3] F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S.M. Figueira, J. Hayes, G. Obertelli, J.M. Schopf, G. Shao, S. Smallen, N.T. Spring, A. Su, and D. Zagorodnov. Adaptive Computing on the Grid using AppLeS. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 14(4):pages 369–382, 2003.

[4] A. Bonner, A. Shrufi, and S. Rozen. LabFlow-1: A Database Benchmark for High-Throughput Workflow Management. In *Advances in Database Technology – EDBT'96, Proceedings of the 5th International Conference on Extending Database Technology*, pages 463–478. Springer, 1996.

[5] S. Chandrasekaran, O. Cooper, A. Deshpande, M. Franklin, J. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2003.

[6] Workflow Management Coalition. Terminology and Glossary. Technical Report, Document Number WFMC-TC-1011, 1999., 1999.

[7] I. Foster, J. Vockler, M. Wilde, and Y. Zhao. Chimera: A Virtual Data System for Representing, Querying, and Automating Data Derivation. In *Proceedings of the 14th International Conference on Scientific and Statistical Database Management*, pages 37–46. IEEE Computer Society, 2002.

[8] E. Gallopoulos, E. Houstis, and J. Rice. Computer as Thinker/Doer: Problem-Solving Environments for Computational Science. *IEEE Computational Science and Engineering*, Vol. 1(2):pages 11–23, June 1994.

[9] L. Golab and T. Ozsu. Issues in Data Stream Management. *SIGMOD Record*, Vol. 32(2):pages 5–14, 2003.

[10] R. Goldman and J. Widom. WSQ/DSQ: A Practical Approach for Combined Querying of Databases and the Web. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 285–296. ACM Press, 2000.

[11] Object Management Group. Object Management Architecture Guide. Technical Report, Revision 3.0, OMG Document 97-5-5, 1997., 1997.

[12] F. Horn, G. Vriend, and F. Cohen. Collecting and Harvesting Biological Data: the GPCRDB and NucleaRDB Information Systems. *Nucleic Acids Research*, Vol. 29:pages 346–349, 2001.

[13] T. Lee, N. Rinaldi, F. Robert, D. Odorn, Z. Bar-Joseph, G. Gerber, N. Hannett, C. Harbison, C. Thompson, I. Simon, J. Zeitlinger, E. Jennings, H. Murray, D. Gordon, B. Ren, J. Wyrick, J.B. Tagne, T. Volkert, E. Fraenkel, D. Gifford, and R. Young. Transcriptional Regulatory Networks in *Saccharomyces cerevisiae*. *Science*, pages pages 799–804, 2002.

[14] M. Miller, C. Hansen, and C. Johnson. The SCIRUun Parallel Scientific Computing Problem Solving Environment. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, 1999.

[15] R. Prodan and T. Fahringer. A Web Service-Based Experiment Management System for the Grid. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, pages 85–96. IEEE Computer Society, 2003.

[16] N. Ramakrishnan, L. Watson, D. Kafura, C. Ribbens, and C. Shaffer. Programming Environments for Multidisciplinary Grid Communities. *Concurrency and Computation: Practice and Experience*, Vol. 14(13–15):pages 1241–1273, 2002.

[17] V. Raman and J. Hellerstein. Partial Results for Online Query Processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'02)*, pages 275–286. ACM Press, 2002.

[18] V. Raman, B. Raman, and J. Hellerstein. Online Dynamic Reordering for Interactive Data Processing. In *Proceedings of International Conference on Very Large Data Bases (VLDB'99)*, pages 709–720, 1999.

[19] R. Stata, K. Bharat, and F. Maghoul. The Term Vector Database: Fast Access to Indexing Terms for Web Pages. *Computer Networks and ISDN Systems*, Vol. 33(1–6):pages 247–255, 2000.

[20] A. Verstak. Data and Computation Modeling for Scientific Problem Solving Environments. Master's thesis, Virginia Tech, 2002.

[21] G. Vossen and M. Weske. The WASA2 Object-Oriented Workflow Management System. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'99)*, pages 587–589. ACM Press, 1999.

[22] M. Weske, G. Vossen, and C. Medeiros. Scientific Workflow Management: WASA Architecture and Applications. Technical Report, Fachbericht Angewandte Mathematik und Informatik 03/96-I, 1996.

[23] J. Widom and S. Ceri. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann, 1996.