University of Alberta

Library Release Form

Name of Author: Steven Bromling

Title of Thesis: Meta-programming with Parallel Design Patterns

Degree: Master of Science

Year this Degree Granted: 2002

. . . .

Steven Bromling
#1010, 10149 Saskatchewan Drive
Edmonton, AB
Canada, T6E 6B6

Date: . . . .

University of Alberta

Meta-programming with Parallel Design Patterns

by

Steven Bromling

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Spring 2002

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Meta-programming with Parallel Design Patterns** submitted by Steven Bromling in partial fulfillment of the requirements for the degree of **Master of Science**.

. . . .
Dr. Jonathan Schaeffer

. . . .
Dr. Duane Szafron

. . . .
Dr. Mike Carbonaro

. . . .
Dr. Paul Lu

Date: . . . .

Acknowledgements

First, I would like to thank my fiancée, Phoebe Jane Elliot, for her love, support and inspiration during the writing of this dissertation, and the research leading up to it.

I would like to extend my gratitude to the CO_2P_3S research team, for their efforts that made this work possible. Steve MacDonald, John Anvik and Kai Tan all proved instrumental in the completion of this MetaCO_2P_3S research. Thanks especially to my supervisors, Jonathan Schaeffer and Duane Szafron, for their guidance and patience.

I would also like to acknowledge the efforts of my thesis committee, for the suggestions that they provided for the improvement of this dissertation.

I would like to thank my family and friends, for their support and understanding, in particular as my attentions were diverted towards this goal.

Lastly, am grateful to the Natural Science and Engineering Research Council of Canada and the Alberta Informatics Circle of Research Excellence for their financial support of this research. In addition, the funding provided by the University of Alberta through a Walter H. Johns Graduate Fellowship, and the Department of Computing Science through a Teaching Assistantship was greatly appreciated.

Abstract

A critical shortcoming of template- or pattern-based systems for parallel programming is their lack of extensibility. Templates are typically limited in number or scope, thereby narrowing the applicability of a given system. This dissertation describes my approach for addressing the extensibility problem in the CO_2P_3S parallel programming system. The tool I developed, called MetaCO_2P_3S, provides the ability for pattern designers to design and add new pattern templates to CO_2P_3S. These pattern templates are equivalent in form and function to those provided by the developers of CO_2P_3S, although stored in a system-independent format. This makes them suitable for storing in a repository to be shared throughout the user community. The validity of MetaCO_2P_3S is illustrated through the creation of new pattern templates.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Computer programs can enjoy potential performance benefits by exploiting parallelism. However, parallel programming is challenging for most developers, due to the issues raised by concurrency. Unfortunately, even though the field of parallel programming is not young, the high-level tool support for parallel programming is still immature. This is partly due to the belief that adding layers of abstraction will adversely affect overall performance.

There are a number of forms that concurrency can take, depending on the requirements of a given algorithm. Finding the appropriate parallel design for a particular program can be a difficult problem. There is an arguable need for tools to assist developers with this problem, in particular for those who are new to parallel programming.

The struggle between the requirements for speed and developer assistance suggests that a compromise is necessary. One approach being taken to meet this compromise is the development of parallel programming environments that employ templates to assist with program design and implementation. One such system, from our research group at the University of Alberta, is called CO_2P_3S[1] [MSSB00a, MSS97]. CO_2P_3S uses design patterns to generate frameworks for parallel programs.

1.1 Motivation

In over twelve years of experience with template-based parallel programming environments, our research group has identified a number of problems that are preventing their widespread acceptance [SSS98]. One of these problems is the rigidity and narrow scope of the environments. Current academic tools only support a small number of templates and, with only a few exceptions, do not allow the creation of new templates. The lack of a rich set of templates is one of the major reasons preventing these tools from moving out of academia and into mainstream use.

[1] Correct Object-Oriented Pattern-based Parallel Programming System, pronounced "cops."

Finding a solution to this problem is not easy. Building a template-based parallel programming system is already a difficult task. Ensuring that the same environment can support the addition of new templates adds further complications. Our community needs a meta-programming tool that enables the creation of new templates and the modification of existing ones. The templates must be generic enough to be usable in a variety of systems. If the templates are defined in a system-independent manner, they can be stored in a central repository. Submissions could be made to the repository from throughout the parallel programming community, and it would serve to remove a major impediment to the acceptance of template-based systems.

1.2 Contributions

This dissertation describes my approach to adding extensibility to template-based parallel programming environments. This undertaking was rife with challenges. First, I needed to determine what information was required to make a complete template description, and how that information could be stored. Then CO_2P_3S had to be abstracted so that it could integrate with template descriptions, including their graphical representations, in a modular fashion. Finally, I needed to specify how parameters could be used to specialise templates by generating different code instances, and how parameter values could be gathered from a user.

My tool for creating new pattern templates is called MetaCO_2P_3S, since it was built as an extension to our CO_2P_3S environment. MetaCO_2P_3S creates first-class pattern templates that integrate seamlessly with the CO_2P_3S environment, since they are identical in form and equivalent in function to the pattern templates supplied with the tool. This dissertation describes the meta-programming approach taken in MetaCO_2P_3S, and extends a challenge to the community to leverage our technology to create a pattern template repository.

1.3 Organisation

Chapter 2 introduces the CO_2P_3S environment, describing its status prior to the research in this dissertation, and its available palette of pattern templates. Further motivation for my meta-programming extension is included.

Chapter 3 discusses the development of the MetaCO_2P_3S tool and how I met my goal of extensibility. This begins with a definition of the required components of a pattern template. It goes on to describe the modifications I made to the CO_2P_3S environment to support modular pattern templates. Finally, the design of the MetaCO_2P_3S tool and its underlying architecture are introduced, complete with a description of the pattern creation process from the pattern designer's perspective.

Chapter 4 shows how I validated the MetaCO_2P_3S tool. As part of this process, I not only recreated all of the patterns in the original CO_2P_3S, but I also added new patterns. As part of this work, MetaCO_2P_3S was used to extend CO_2P_3S from its target shared-memory parallel platform to distributed memory environments on networks of workstations.

In Chapter 5, I promote the need for a pattern repository. We also demonstrate the generality of my meta-programming approach, showing how generic patterns can be created using MetaCO_2P_3S. This finding extends the power of CO_2P_3S and MetaCO_2P_3S beyond the domain of parallel programming.

Chapter 6 describes research related to MetaCO_2P_3S, and Chapter 7 summarises the contributions of this dissertation, introducing ongoing enhancements to CO_2P_3S and describing directions for future work.

Appendix A gives instructions for the installation of CO_2P_3S and MetaCO_2P_3S. Examples of the system-independent file formats for pattern templates, as introduced in this research, are given in Appendix B.

Chapter 2

An Introduction to CO_2P_3S

Our parallel programming system, called CO_2P_3S, uses object-oriented programming techniques to simplify parallel programming [Mac01, MSSB00a, MSSB00b]. This research follows the previous parallel programming system from our group, called Enterprise [SSLP93], which evolved from an earlier prototype called FrameWorks [SSG89]. Enterprise used a business model as an abstraction for the communication patterns in programs for networks of workstations. The environment included many supporting tools, such as detailed debuggers and animated playback mechanisms. The research goal for both Enterprise and CO_2P_3S was to provide a means for software developers to parallelise and speed up their sequential programs with little effort, while minimising the possibility of new errors being introduced. Using CO_2P_3S, developers can create, compile and execute programs for concurrent architectures. This chapter describes the state of CO_2P_3S prior to the meta-programming extension, and motivates the work done in this dissertation.

In CO_2P_3S, developers identify parallel design patterns that describe their application's basic structure, parameterise the patterns to specialise them for their needs, and implement application-specific code within the automatically generated framework that hides the entire parallel infrastructure. CO_2P_3S targets programmers looking for reasonable speedups in their sequential applications in return for a modest programming effort.

A *parallel design pattern template*, or *pattern template* for short, consists of two main components. The first is a parallel design pattern, and is described in Section 2.1. The second is a set of framework implementations representing the various forms that the design pattern can take. The transition process from design patterns to frameworks is described in Section 2.1.1.

There is currently no support in the CO_2P_3S environment for the design pattern selection process. The onus is on the developer to deduce the appropriate pattern templates for their application. It is possible that this process can be automated in the future, based on the results of pattern language research [MMS00].

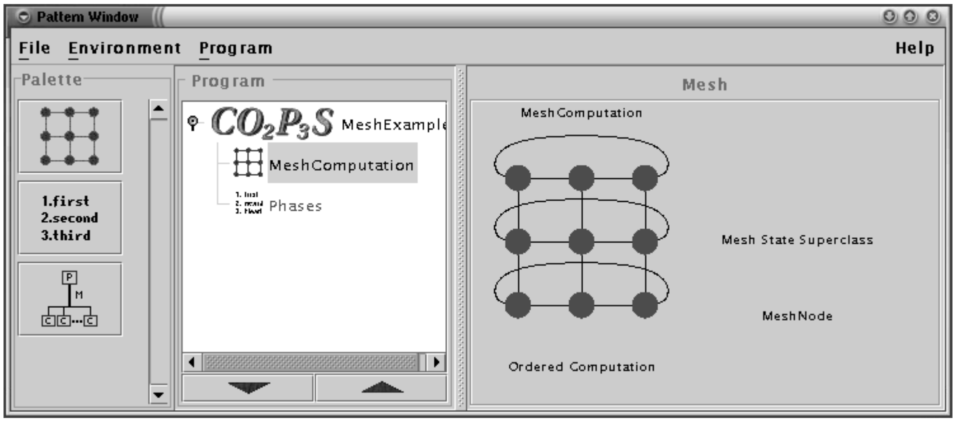


Figure 2.1: A new CO_2P_3S application.

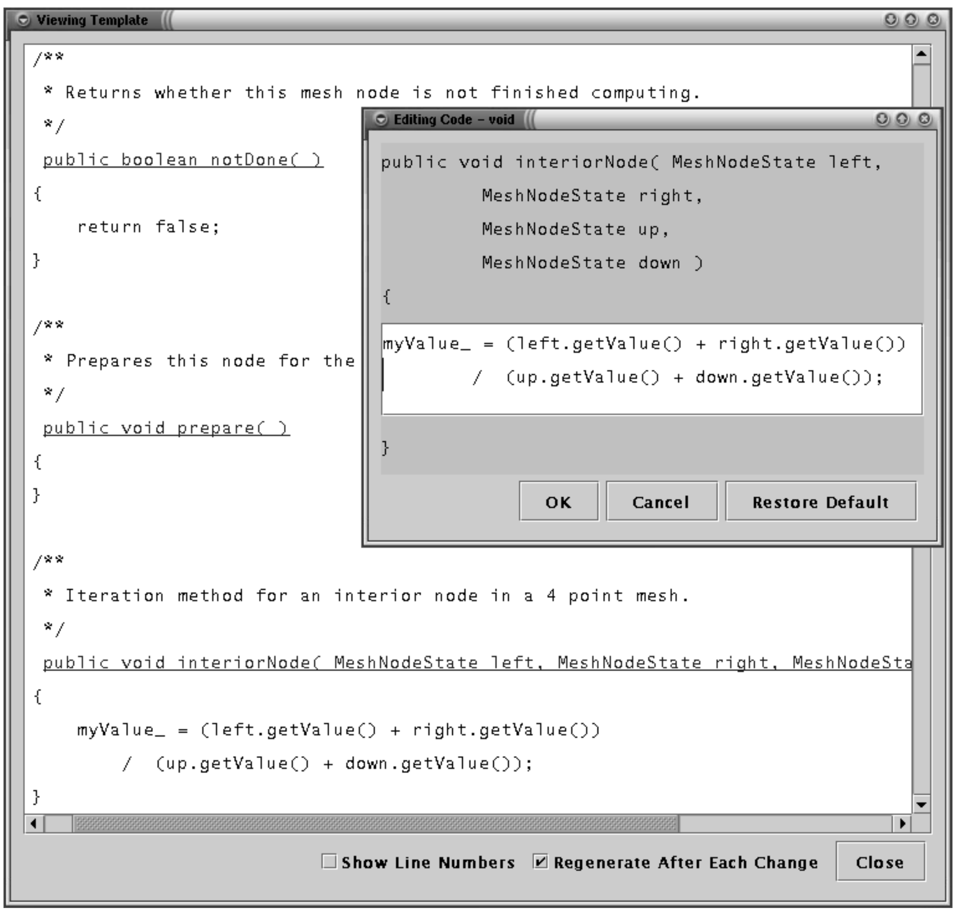


Figure 2.2: The user's view into the framework generated by CO_2P_3S.

One of the key features of CO_2P_3S is its separation of system-generated parallel code and user-provided sequential code. Parallel code sections are typically the most complicated part of an application. CO_2P_3S hides them from the user in the frameworks it generates. This separation helps to maintain correctness in applications, and it greatly simplifies users' implementation efforts. We ensure that the separation does not limit CO_2P_3S users by providing a programming model that has three layers of abstraction. The highest layer does not provide any access to parallel code. In the intermediate layer, a high-level explicitly parallel programming language enables manipulation of the parallel structure. At the lowest layer, CO_2P_3S provides native object-oriented code for the entire application.

Figure 2.1 shows CO_2P_3S with a user application involving two design pattern templates. The pattern template selected in the middle segment of the user interface and displayed in the pattern window on the right represents a two-dimensional parallel mesh computation, which supports iterative computations across a surface. Using the pattern window, a CO_2P_3S user can customise the pattern template instance by setting class names for the framework and parameterising the pattern template so that it matches their application. One of the parameters available for the Mesh pattern template is the boundary condition that allows the mesh to be a rectangular surface, a cylinder or a torus. Another parameter switches the number of neighbours for the nodes in the mesh from four to eight. After customising the pattern template, the user requests that the appropriate framework code be generated. The current pattern templates in CO_2P_3S are configured to generate shared-memory Java code. Figure 2.2 illustrates the user's view of the framework, showing a class that represents a single node in the mesh structure. Users are not allowed to edit the class in this window. This prevents them from modifying method or class signatures. Instead, hyper-links are provided for user-modifiable locations in the code. The inset window is the result of following one such hyper-link, and allows the user to enter the selected method's body.

2.1 Parallel Design Patterns

Design patterns, as popularised by Gamma, Helm, Johnson and Vlissides [GHJV95], capture and document recurring problems in object-oriented software design. Necessary components of a design pattern document include both a description of a problem and its solution. The problem description incorporates a discussion of the pattern's applicability. There should be sufficient detail in the solution for a developer to fully complete the associated aspect of their design.

As an innovative subset of the more general realm, parallel design patterns deal with problems related to concurrent programming. There are many difficulties that make parallel programming a daunting endeavour for developers, including:

- the identification of the parallelism in a given application, if any.
- the construction of the parallelism, ensuring that the threads of execution synchronise at key points and transfer data between threads in a timely fashion.
- the target architecture (whether it be a distributed network of workstations or a shared memory multi-processor) can affect the implementation language and the algorithms used.
- errors or omissions can cause the program to function erratically and incorrectly, making it difficult to debug.
- various performance enhancements may be required to attain reasonable program speedups, including increasing task granularity, balancing the load between threads of execution and reducing communication.
- the application may need to scale to machines with different capabilities, or be portable across platforms.

The cumulative time for all of these activities can make the cost of developing parallel programs prohibitive. Traditionally, the onus was almost entirely on the developer to deal with these issues. A design pattern catalogue can help by not only identifying the type of parallelism that best captures their problem, but also by prescribing an appropriate solution. An example parallel design pattern catalogue is included in [Mac01].

2.1.1 From Design Patterns to Frameworks

Every design pattern includes an abstract solution to a particular design problem. The solution is general, to avoid dependencies on a particular architecture or programming language. If this restriction is relaxed, the result is no longer a design pattern, by definition. Our alternative is to use design patterns to generate code skeletons, which can serve as the basis for a collection of different programs.

A framework embodies the intent of a particular design pattern. It provides a set of abstract classes, and is customised for a specific domain through the implementation of one or more subclasses. The application structure is provided by the framework, and is not modified by the developer. Because of this, a properly constructed framework can guarantee structural correctness. In the domain of parallel programming this is particularly useful, since a framework can encapsulate the synchronisation and communication needed for concurrency, freeing the developer from a difficult part of their application.

A small complication arises in the conversion process from design patterns to frameworks because of the possibility of design pattern parameters. Often there are slight variations in a design pattern that affect its implementation. These variations, or parameters, remove the possibility of a one-to-one mapping between design patterns and frameworks. Instead, many different frameworks may be required to provide all possible parameterisations of a given design pattern. A parameter in a design pattern does not correspond to a single argument in a framework method. In fact, a parameter choice at the design pattern level may result in the existence or absence of a set of methods or portions of method bodies in the generated framework. Equipping developers for this situation requires a system that automatically generates correct frameworks from a selected design pattern with the appropriate parameterisation. This is one of the roles of the CO_2P_3S system, as discussed in [MSSB00a] and [MSS97].

2.2 The Original CO_2P_3S Implementation

A system that automatically generates framework code from design patterns can be built on any combination of architecture and programming language, provided support is available for parallelism. Early in the development of CO_2P_3S, Java was selected as the programming language, and shared-memory machines as the parallel environment. That decision has provided some degree of architectural independence, allowing CO_2P_3S to be run on many different flavours of UNIX, from dual-processor Linux machines to large SGI Origin 2000 supercomputers. Parallelism on these shared-memory systems is achieved through Java native threads. The object-oriented capabilities of Java are needed for the generated frameworks. It should be stressed that the design of CO_2P_3S does not preclude the use of other programming languages or architectures (e.g. C++ on a distributed network of workstations).

CO_2P_3S has been in development for a number of years now. It began with a code generator, which given a design pattern and a specific parameterisation as input, automatically produced framework code. There was no user interface, and therefore no easy way to view the pattern catalogue or provide the parameterisation for a selected pattern.

As a summer research assistant, I implemented a graphical user interface (GUI) for CO_2P_3S [MSSB00b], also in Java. The main purpose of the GUI is to gather design pattern parameter information, and interface with the code generator to create the appropriate framework code. Development efforts attempted to ensure that the GUI representation of pattern templates was modular, and thus both maintainable and extensible. Figure 2.3 shows the three functional areas of the main GUI window. The inner window, entitled Example, contains the start of a parallel application.

- On the left, the Palette displays iconic representations of the parallel pattern

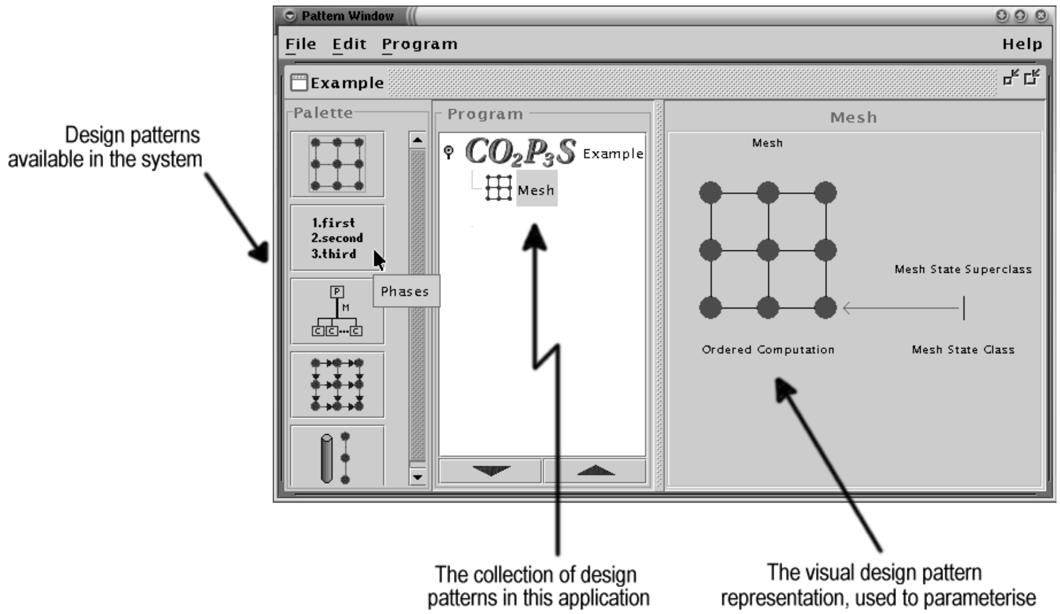


Figure 2.3: The original CO_2P_3S GUI.

templates currently in the catalogue. As the figure shows, rolling the mouse over one of the icons displays the name of the corresponding pattern. In addition, by right-clicking the mouse on an icon, one can choose to view the associated design pattern document.

- The middle Program panel contains the list of pattern templates selected by the developer for inclusion in their application. In this case, only the Mesh pattern is being used. However, it is possible to combine more than one pattern when building an application.

- The segment on the right is where the developer parameterises a pattern template, and sets the class names to be used in the framework.

The original CO_2P_3S is the result of a large implementation effort. The GUI code consists of 56 Java classes with an NCSS[1] line count of 5,844. Adding the framework creation code and three design patterns totals 89 Java classes with an NCSS line count of 7,762.

Section 2.3 provides further discussion related to the CO_2P_3S implementation, by describing typical system usage. Section 2.4 describes three design patterns that were implemented in the original CO_2P_3S.

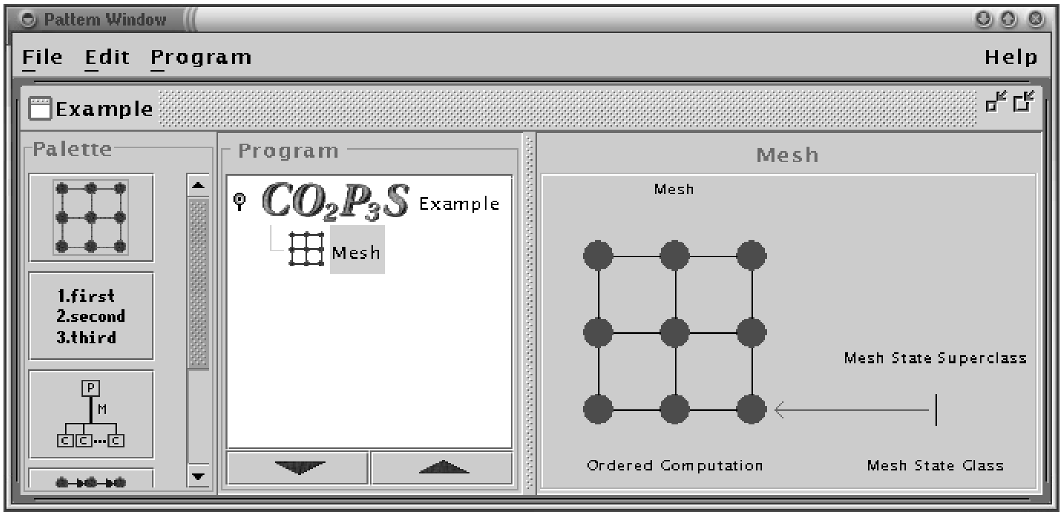


Figure 2.4: A new application with one pattern template instance.

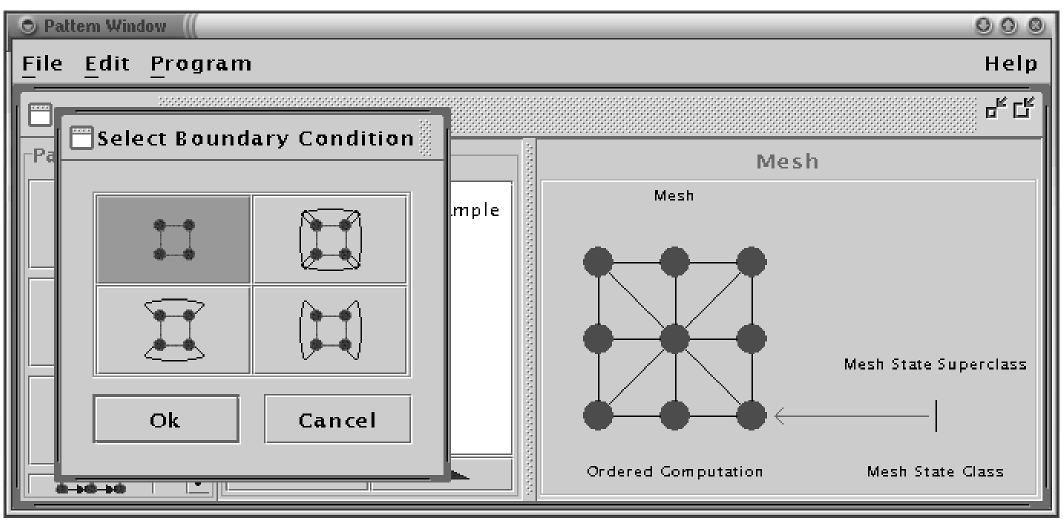


Figure 2.5: Parameterisation of the pattern template.

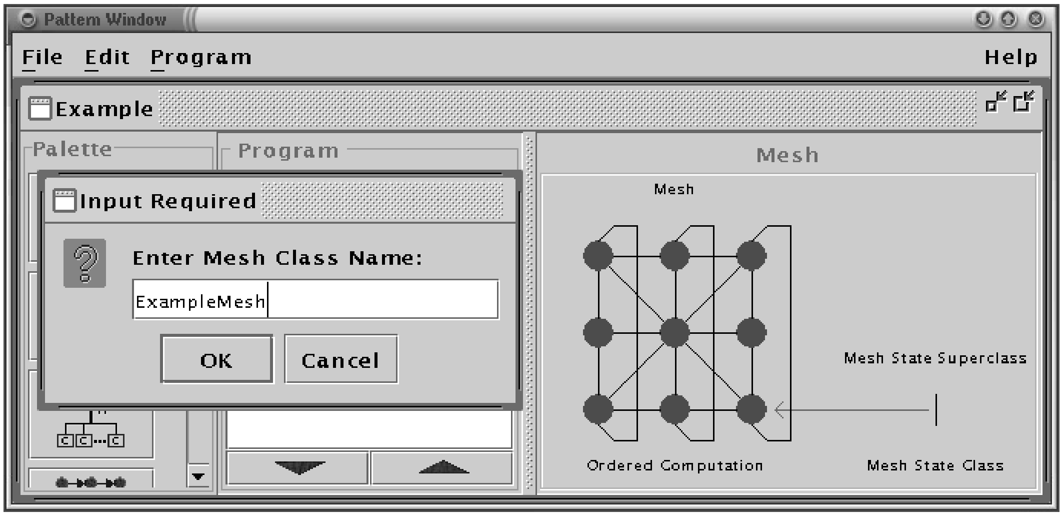


Figure 2.6: Setting a framework class name.

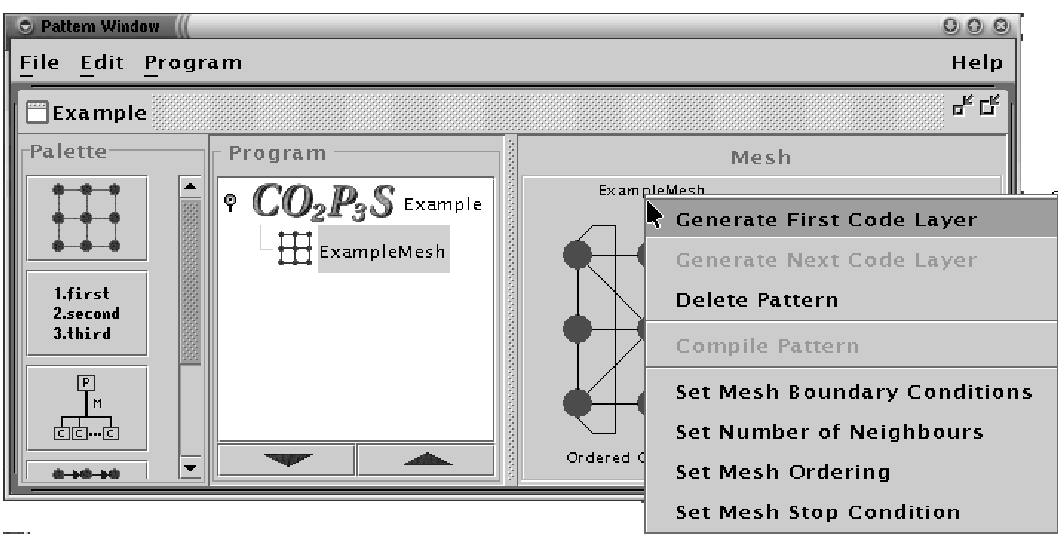


Figure 2.7: Generating the framework code.

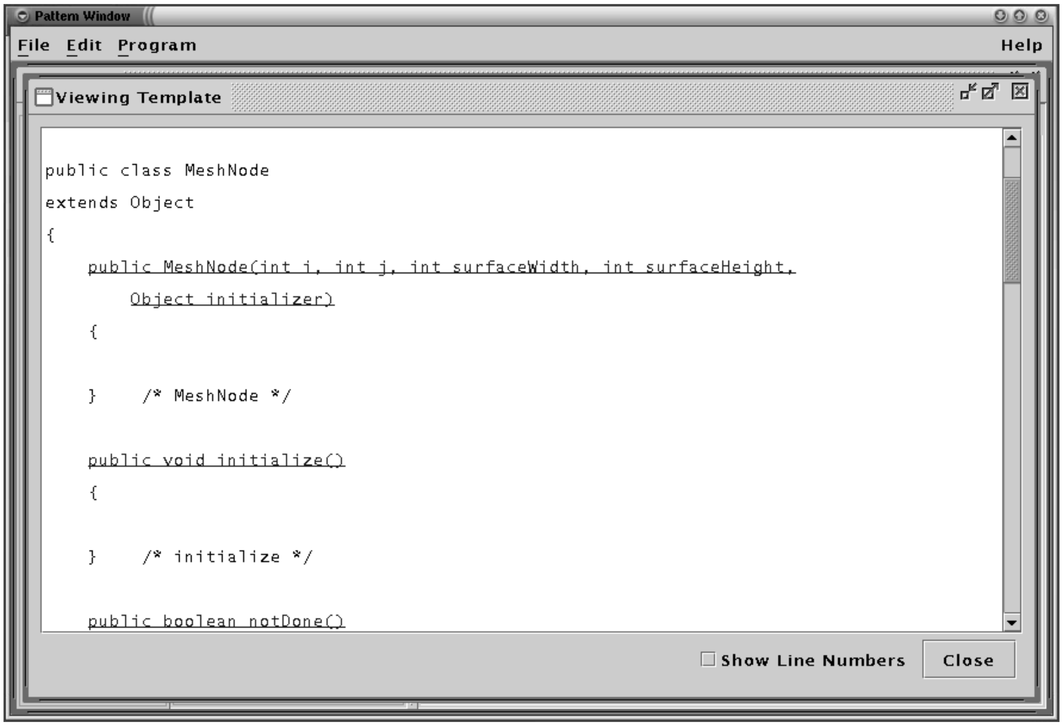


Figure 2.8: Viewing the framework code in CO_2P_3S.

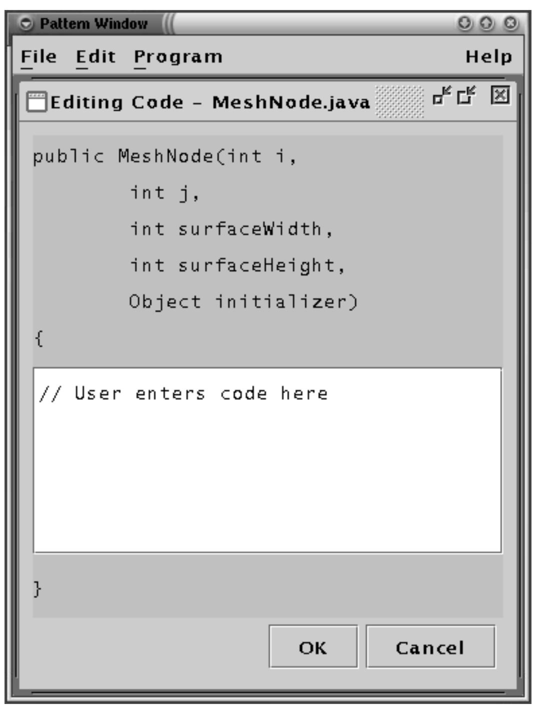


Figure 2.9: Editing a framework method.

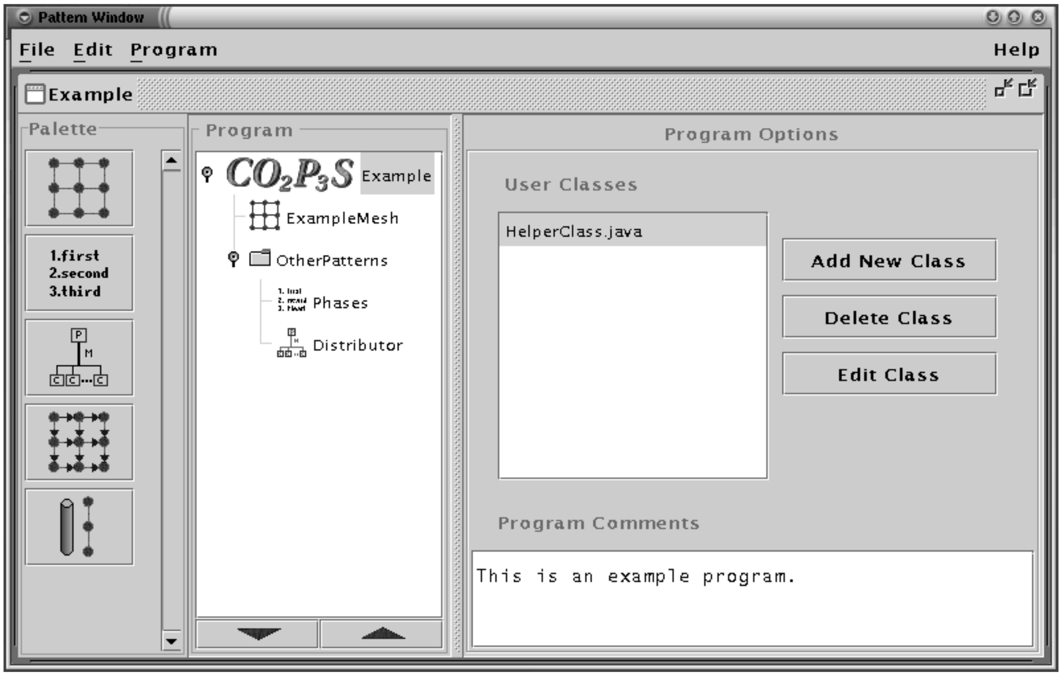


Figure 2.10: Setting the application options.

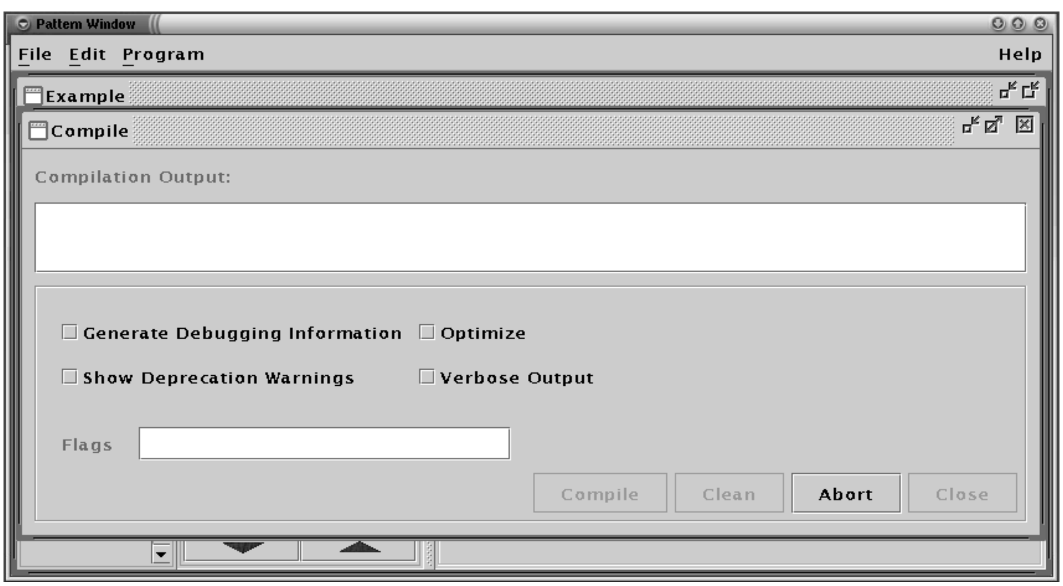


Figure 2.11: Compiling the parallel application.

2.3 CO_2P_3S Usage

Figures 2.4 to 2.11 step us through a typical usage of the CO_2P_3S GUI. A more detailed usage description of CO_2P_3S, and the Mesh pattern in particular, is in [MSSB00b]. In Figure 2.4, a single Mesh[2] parallel pattern template instance has been added to a new application. Prior to this, the CO_2P_3S user would have spent time analysing their application to identify the pattern template that most aptly described their parallel design problem.

The parameter affecting the mesh boundary condition is being set through a graphical dialog in Figure 2.5. The name for one of the framework classes is being set in Figure 2.6. These two examples demonstrate the ease with which a pattern template can be parameterised in the CO_2P_3S GUI.

Figures 2.7 through 2.9 show the process of creating and editing the framework code. The template viewer only allows the user to edit the code of certain methods, called the framework hook methods, thereby preventing accidental modification of method signatures, or other damaging changes. The user enters only sequential code at these hooks, as the parallelism is hidden by the framework.

An application in which a user has composed multiple pattern templates is shown in Figure 2.10. This figure also shows the capability for adding external user classes to an application. Once all of the necessary pattern templates for an application have been added, parameterised, and had their framework code templates completed, the application can be compiled and run, as in Figure 2.11. It should be noted that an effort was made during the development of each of the CO_2P_3S pattern templates to always generate framework code that compiles without any modifications. This allows developers to incrementally modify and test their pattern instances.

One of the important features surfacing early in the design of CO_2P_3S was a three layered approach to parallel program development. Thus far, this section has discussed only the highest level of abstraction, called the Patterns Layer. This layer includes the concept of framework code, as generated by parameterised parallel pattern templates. User-supplied hook methods are used to create a complete application. This layer also provides a guarantee of correctness, by ensuring that users have no effect on the constructs of concurrency in their programs.

Two further abstraction layers were described in [MSS97]. They are called the Intermediate Code Layer, and the Native Code Layer. Both are meant to increase the openness of CO_2P_3S applications by allowing developers to gain access to the framework code, in two different formats, for the purpose of performance tuning. The framework code was designed for safety, to prevent the possibility of errors. How-

[1]Non-Commenting Source Statements, approximately equivalent to the number of ";" and "{" characters in Java source code.

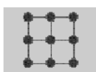
[2]The Mesh pattern is described in Section 2.4.1.

ever, some applications may not require all the safety features built into a particular framework, and it is important to allow the developer to make modifications. At the Intermediate layer, abstract concurrent constructs are represented in a high-level format. The Native layer removes the parallel abstractions, and provides access to the entire object-oriented code-base of the framework.

2.4 Available Parallel Design Pattern Templates

Prior to the start of my research, only three fully implemented pattern templates existed in CO_2P_3S. The implementation effort required to add new pattern templates was too high to justify adding more, as will be discussed in Section 2.6. The three patterns are briefly described below. A more in-depth discussion of these patterns can be found in [MSS99] and [Mac01].

2.4.1 The Mesh Pattern Template

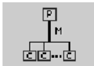


The Mesh pattern template supports surface mesh computations. General mesh computations are not supported by the pattern, which instead focuses on regular meshes consisting of $n \times m$ points on a two-dimensional surface. In a mesh computation, a sequence of steps affects each point on the surface. Each point changes state based on its previous state, and that of each of its neighbours. The computation typically concludes when all of the mesh points have reached some final agreed upon state. The pattern template solution ensures that each iteration, or step, affects every point in the mesh before the following iteration begins. This task is made more difficult by the fact that different points may have been computed by different threads of execution.

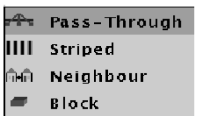
The pattern template user does not have to worry about the issues of concurrency. Instead, they only provide a termination condition, and write code that describes how a mesh point computes its state, based on the provided states of each neighbour.

In addition to specifying class names for the generated framework code, the Mesh pattern template user must specify two parameters. The first parameter determines whether the boundaries of the mesh have edges connecting them to neighbours on the opposite edge. This means that there can be meshes that are fully toroidal, horizontal-toroidal, vertical-toroidal or non-toroidal. The second parameter sets the number of neighbours for each point in the mesh. There can be either four neighbours (up, down, left and right), or eight (also includes the diagonals). The size of the mesh is not required during the design phase, as it is provided as a run-time parameter to the Mesh framework.

2.4.2 The Distributor Pattern Template



The Distributor pattern template provides a form of data parallelism. It divides the execution of methods that contain arrays of data in their arguments, and delegates computation for each array segment to a different child thread. Methods that have too fine a granularity to allow efficient parallelisation can be run sequentially. The Distributor pattern user must provide a list of methods to be executed. For each method that can be parallelised (i.e. has an argument which is a one-dimensional array of elements), they must specify the distribution amongst the children. The valid distributions are:



Pass-Through the entire array is passed to each child.

Striped child i (of n children) receives the array elements at $(i, i+n, i+2n, ...)$.

Neighbour child i receives elements i and $i+1$ from the input array.

Block n contiguous sections of the array are distributed amongst the n children.

2.4.3 The Phases Pattern Template

The Phases pattern template, also referred to as Method Sequence, is a special pattern that provides no concurrency. Instead, the Phases pattern can act as the glue between other pattern templates in an application. It takes a list of methods as its only parameter, and is responsible for the sequentially ordered execution of those methods.

2.5 Case Study: Image Processing

Prior to the commencement of this dissertation research, I performed a study on the usability and performance of the CO_2P_3S environment. The study used the following set of simple image processing algorithms:

Contrast Stretching: Each pixel in an image that falls within a given input range is stretched to fit a given output range.

Sobel Edge Detection: Applies a mask that filters each pixel in a given image, creating an output image that highlights the "edges," or areas of rapid frequency change.

Median Noise Reduction: Reduces the number of "noisy" pixels in a given image by applying a simple mask to each pixel.

Each of the algorithms was implemented three different ways using Java. The first implementation was a simple sequential solution. The second was a hand-coded parallel implementation. The third implementation used the CO_2P_3S environment.

The only parallel design pattern template that was available in CO_2P_3S and could be applied to the image processing problems was the Mesh. Unfortunately, this selection was not ideal. One reason for this was that although the Mesh pattern template was designed to operate on flat $n \times m$ surfaces, it was meant to be used for iterative processes that repeat a computation at each element over time. This is in contrast to the image processing operations, which computes only once at each pixel, or surface location.

As might be expected, the performance numbers achieved in this study were disappointing. However, a positive result surfaced from this work. In determining that the Mesh pattern template was inappropriate for the selected image processing algorithms, we concluded that a new pattern template would be required in the CO_2P_3S environment. An implementation for this pattern template could emulate my hand-written parallel solution, which segmented a given image into n equal blocks and then ran n parallel threads of execution to process it. Our need for a new pattern template further motivated and supported my extensibility research.

2.6 Adding Parallel Design Pattern Templates

The small parallel pattern template catalogue was a severe limitation for CO_2P_3S. If a developer's application required a pattern template that was not provided by CO_2P_3S, the system was rendered virtually unusable to them. Thus, it was crucial for CO_2P_3S to allow new pattern templates to be added easily. The GUI was built in a modular way to allow for this necessity. However, a large implementation effort was still required to add each new pattern template. Each pattern addition consisted of two stages.

The first step was the implementation of a plug-in GUI module. This required sub-classing an abstract module that provided assistive functionality. Next, a visual representation of the pattern had to be developed. This included parameter settings and framework class names. Certain parameters needed tailored dialogues for retrieving user requirements. Action handlers had to be written, and the pattern needed to have a method to capture a run-time snapshot of itself for permanent storage.

In addition to the GUI module, a framework generator interface was required. This module was responsible for gathering data from the GUI component at the appropriate time and packaging it in a format usable by the code generator. Template code for each of the framework classes had to be provided, with optional code for each of the pattern template parameters. The code generator needed to be instructed to create the framework based upon the provided parameterisation.

Finally, after implementing and testing both of these modules, the new pattern template could be added to the GUI by adding its name to the CO_2P_3S configuration file.

2.6.1 Motivation

As a means for the CO_2P_3S internal development team to add new pattern templates, the status quo was sufficient, but unpleasant. It stalled the addition of a number of known pattern templates, and limited the size of the provided catalogue. Furthermore, it was thought that advanced CO_2P_3S users should be given the ability to add patterns themselves, since the provided catalogue would never be complete. It became apparent that a simpler procedure for adding pattern templates was required to make CO_2P_3S a viable system for parallel programming.

With these thoughts in mind, the aim of my research was to realise the goal of extensibility in CO_2P_3S by creating a meta-programming tool for adding new parallel pattern templates into the system. To achieve this goal, I needed to normalise the representation of pattern templates, and create a simplified high-level abstraction for pattern template creation. I then needed to design and implement a tool to support this abstraction.

Chapter 3

A Meta-programming Tool for CO_2P_3S

The previous chapters motivated the need for extending our pattern-based parallel programming system by providing the ability to add new templates. They described the lack of extensibility as a major impediment to the acceptance of template-based parallel programming environments. If an application cannot be implemented using a given programming environment, it calls into question the utility of that tool. Programmers are unlikely to invest effort learning an environment that may not meet their needs in the future.

My hypothesis is that not only is there a need for a large number and variety of pattern templates to cover all parallel programming needs, but also that there are still undiscovered parallel design patterns. Even if posterity proves me wrong, and demonstrates that only a handful of complete pattern templates suffice to cover the domain of parallel programming, MetaCO_2P_3S will still be regarded as a useful tool. Pattern templates are seldom complete after their first design iteration, and MetaCO_2P_3S makes the task of pattern modification much simpler.

To address the extensibility problem, I have created a tool that allows parallel and object-oriented programming experts, called *pattern designers*, to create new pattern templates. The new pattern templates are first-class, meaning they are indistinguishable in form and equivalent in function to the pattern templates included with CO_2P_3S. Analogous to the manner in which CO_2P_3S makes it easier to write parallel programs using pattern templates, MetaCO_2P_3S makes it easier to write pattern templates for CO_2P_3S.

This chapter describes the extensibility research. Section 3.1 enumerates the necessary components for a parallel design pattern template. Section 3.2 describes the modifications required in CO_2P_3S to support modular pattern templates. Sections 3.3 through 3.6 describe the design of MetaCO_2P_3S, and how it is used to create new pattern templates. Finally, Sections 3.7 and 3.8 describe the process of testing pattern templates and importing them into the CO_2P_3S environment.

3.1 The Necessary Components of a Pattern Template

Before embarking upon the development of the MetaCO$_2$P$_3$S tool, it was important for us to identify and define the necessary components of a pattern template. As part of this investigation, I needed to assess the possible differences between pattern templates to ensure that my definition was complete. The three parallel pattern templates in the original CO$_2$P$_3$S implementation were my primary source of inspiration for this task.

The entire skeleton of a pattern template, as described below, was defined at the outset of my research. However, a number of the details, particularly to do with the implementation, were determined later or arose through trial and error.

I have left the responsibility of identifying new design patterns to the pattern designer. This task involves isolating newly-discovered recurring patterns and the various forms that they can take based on pattern parameters, then creating a framework that hides the parallelism details. Designers should note the aspects of their frameworks that are affected by different parameter settings.

Neither CO$_2$P$_3$S nor MetaCO$_2$P$_3$S were built to support design pattern discovery, since it is a hard problem that has yet to be solved. One example of research that relates to the discovery problem is the PatternLint tool [SSC96], which checks a program to ensure that it follows the design pattern contracts that were specified as part of its design. However, PatternLint does not discover known design patterns in an application. Furthermore, even if PatternLint could be extended to discover the design patterns in a program, it would still be unable to identify unknown design patterns. This last problem is one of many that would need to be solved before automation of the pattern discovery process could be realised.

The rest of this section introduces the components that I identified as being necessary for a pattern template.

3.1.1 Pattern Template Identification and Documentation

Two key features of every pattern template are its name and its documentation. As with generic design patterns, the name gives developers the ability to converse about the given problem and its recommended solution. It is important that the chosen name be memorable, and indicative of the pattern's role. This will also help pattern users with the selection process.

I have extended the naming requirement for pattern templates to include an iconic representation. The chosen icon will represent the pattern template in a graphical user interface. It is important that the icon is useful to pattern template users.

The pattern documentation is at least as important as the name. Like a design pattern document, it describes the problem being solved and the form that the solution takes. Pattern template documentation also includes usage information. The documentation must suffice to serve as the basis for a developer's selection of that pattern for their application.

3.1.2 Class Names

A pattern template must include each of the class names that exist in its framework instances. The template must use generic placeholder names for each of the classes, which define their function in the framework, yet can be replaced by user-supplied values in a particular framework instance. This condition is necessary to allow multiple copies of the same pattern template to be instantiated in a single program. The pattern user will be required to supply at least one class name for each pattern template instance. The remainder of the placeholder class names, necessary to complete the implementation of the design pattern, can be generated simply by adding suffixes or prefixes to the user-supplied class name.

3.1.3 Parameters

The parameters that act on a pattern template allow it to have different behaviours, in order to match a user's requirements. Every possible combination of parameter settings causes a different framework instance to be generated. I have chosen to have three parameter types, and each is described below:

Basic Parameters

Basic parameters cover most common parameter usage cases. They are comprised of either an arbitrary string value, or an enumerated list of choices that must be supplied by the pattern designer. The set of possible configurations range from boolean switches to more elaborate list choices. Pattern users are required to select one and only one value for a given parameter, although a default value can be supplied by the pattern designer. In the Mesh pattern template, introduced in Section 2.4.1, only basic parameters appear. One example is the *number of neighbours* parameter, which can be set to either *four* or *eight*.

Extended Parameters

Extended parameters deal with the relatively uncommon case in which parameter values are in an arbitrary form. Since extended parameters must deal with cases that cannot be covered by basic parameters, extra work is required of the pattern designer. For each extended parameter, the designer must provide a way for users to specify the parameter's value, and the manner in which the given value will affect framework code generation. The Distributor pattern template, introduced in Section 2.4.2, uses a list parameter (see below) composed of extended parameters.

Each entry in the list is a method signature with additional information that alters the distribution of methods with array arguments.

List Parameters

List parameters are a useful subset of extended parameters. They deal with situations in which a pattern template user needs to supply a list of values. The list values can range in complexity from simple strings to complicated extended parameters. As with extended parameters, the pattern designer is required to specify the manner in which a list parameter setting affects framework code generation. However, support is provided for gathering the list values from the pattern template user, and iterating through the list during code generation. The Phases pattern template, introduced in Section 2.4.3, uses a list parameter to gather method names from the user.

3.1.4 GUI Configuration

Pattern templates must integrate with programming environments that have graphical user interfaces, or GUIs. Therefore, it is important that they enable users to visualise their parameter settings through a graphical representation. As an example, Figure 2.1 depicts the Mesh pattern template in the CO_2P_3S environment. Images and textual data are combined to inform the pattern user of the pattern template settings.

3.1.5 User Interaction Capabilities

Since pattern templates are used in GUIs, they must handle user interaction. Pattern users need to set class names and parameter values. They must also be able to generate a framework instance, and populate it with their application code. All of these operations must be dealt with by the pattern template, for example by presenting a dialog of choices to the user.

3.1.6 State Maintenance

The pattern template must be capable of maintaining the run-time state of any of its instances. In addition to the parameter and class name settings, the state includes the programming abstraction layer currently being accessed by the pattern user. The state information must be serialisable to a string, so that a user's programming session can be saved to disk or transmitted across a network.

3.1.7 The Framework Template

In order to generate different framework code for each of its possible parameterisations, a pattern template must include a framework template. A framework template consists of the set of classes that are in each of the framework instances. These classes contain normal Java code, and include additional meta-programming information that indicates the effect of parameters on certain sections.

3.2 Modifying CO_2P_3S to Support Modular Pattern Templates

CO_2P_3S was originally designed with future extensibility in mind, and our research group had the foresight to plan for the continual addition of pattern templates to the system. However, little time was spent optimising the modularity of pattern templates or automating the template creation process. Therefore, the task of creating and adding a pattern template required a significant amount of time and programming effort. One of the first research tasks, after my identification of the pattern template components, was to modify the CO_2P_3S environment to support plug-in pattern template modules.

3.2.1 Decoupling the Framework Template and Pattern Description Components

The first step was to decouple the GUI representation and user interaction components of pattern templates from the framework code generation aspects. We made this choice because of the high degree of similarity across pattern descriptions, and the independence of the framework templates from this similarity. The data flow between these two components now consists only of textual data representing the class name and parameter settings gathered from the pattern user.

This decoupling of the two primary components of pattern templates had a number of advantages. Foremost among these was the ability it gave for us to concentrate on the implementation of each part in isolation. Furthermore, since the pattern descriptions were already closely tied to the CO_2P_3S environment, and the framework generation was a separate component, the decoupling married nicely with the realities of the CO_2P_3S implementation.

3.2.2 Supporting the Pattern Description Component in CO_2P_3S

Since the CO_2P_3S GUI was written in Java, the obvious choice for the implementation language of the pattern description component was also Java. This choice allowed us to use object-oriented abstractions in the design of the plug-in GUI module. The similarities between components are enforced by making their implementations subclasses of an abstract class called PatternPane, which is supplied with CO_2P_3S. The abstract class provides implementations to support each of the following aspects of a pattern template:

Identification: the PatternPane stores the name of the pattern template, and the icons that represent it in the CO_2P_3S environment.

Documentation: the PatternPane contains links to the associated pattern template documentation.

Class Names: with the help of the PatternClass class, the PatternPane stores each of the run-time class names for a pattern template, and supplies a dialog that gathers this information from the pattern user. The PatternPane also provides operations that automatically verify the correctness of a given class name and prevent name clashes with other pattern templates in an application.

Parameters: with the help of the PatternParameter class, the PatternPane stores the user-supplied parameterisation of a pattern template. Dialogs are provided to support gathering the values of basic parameters and list parameters.

Graphical Display: the PatternPane has operations that allow for the easy addition of images and text to the CO_2P_3S GUI display. The layout and display of these elements is handled by the PatternPane. Also, if any images or text values are supposed to dynamically change to represent the current parameterisation of a pattern template instance, these changes are handled by the PatternPane.

User Interaction: in addition to the dialogs provided for class name and parameter value input, the PatternPane provides menu options that enable operations such as framework code generation. It also provides windows for viewing framework code and editing the framework hook methods.

State: the PatternPane maintains all of the state information for a pattern template instance, and provides operations that allow it to be serialised for long-term disk storage or network transfer.

The PatternPane abstract class was part of the original CO_2P_3S implementation, but its scope was much smaller than that of the current incarnation. Subclasses of the new PatternPane class are short and simple, consisting primarily of initialisation code.

3.2.3 Supporting the Framework Template Component in CO_2P_3S

Due to its decoupling from the pattern description, research on framework templates was able to proceed independently. As such, it was not until after the updates to CO_2P_3S and the creation of MetaCO_2P_3S that my attention turned to this problem. The macro language that had been used for code generation in the original CO_2P_3S

environment had been powerful enough to support our needs, but was rather unwieldy, making framework template creation and maintenance difficult. In Section 3.4, the design and architecture of my new framework template is described. The only changes to CO_2P_3S required to support the new format were updates to the PatternPane to allow parameter and class name settings to be written to a text file for use during code generation.

3.3 The Design of MetaCO_2P_3S

At the outset, my research plan was to create an integrated meta-programming tool that would automate as much of the pattern template creation process as possible. Unfortunately, it turned out that a complete solution was beyond the scope of this dissertation. MetaCO_2P_3S deals completely with the pattern description portion, but the planned tool support for the framework template component is not complete.

The MetaCO_2P_3S tool allows a pattern designer to create a new pattern template or modify an existing one. Since I defined a standardised format for pattern templates, the information required of the pattern designer is well-formed and minimal. I strove to ensure that pattern designers were not required to supply anything beyond GUI configuration and the names and types of class names and parameters.

Another design choice was to store pattern template information in a system-independent format. Since our CO_2P_3S environment requires plug-in Java modules of a particular format, this meant that I needed to come up with an intermediate storage representation. My purpose behind this approach was to allow pattern templates to be shared not only amongst CO_2P_3S users, but also throughout a broader community, as more parallel programming environments begin to support the format. Chapter 5 discusses the usefulness of pattern template sharing. Section 3.4 describes the system-independence.

In my current implementation, pattern designers must write the Java code for their framework templates, complete with the meta-programming information for parameterisation. However, I designed this process to be done in a tool that automates adding the meta-programming information.

Because of my choice of Java as the framework template language, the pattern templates generated are completely system-independent. The use of Java reflects only my design choice, and is not a necessity. The generic pattern descriptions could, in fact, be used with framework templates in different languages. The unfortunate aspect of this approach would be that it would separate pattern templates into different categories, based on their implementation languages.

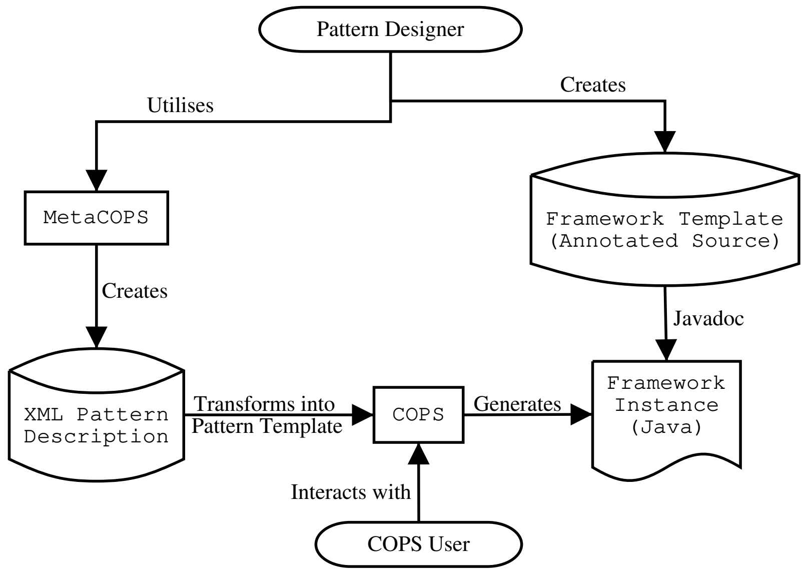


Figure 3.1: The pattern template architecture.

3.4 The Pattern Template Architecture

This section discusses the architecture of the pattern template creation process. I describe how my approach minimises the amount of user interaction required, and show how my implementation is hidden from the user. Figure 3.1 gives an overview of the architecture. My decoupling of the pattern description and framework template components is clearly shown.

Figure 3.2 briefly describes the files used in the CO_2P_3S environment, and the interactions between them. At the top, the creation of the pattern template is depicted. The pattern designer uses MetaCO_2P_3S to generate an XML pattern description, and manually creates the framework template files. At the bottom, the two roles of the pattern user are shown. The first, importing a pattern description into CO_2P_3S, takes an XML file as input, generates a plug-in Java module, and inserts it into the palette of the CO_2P_3S GUI. This step only needs to be done once, as the pattern template becomes part of the user's CO_2P_3S environment. The second role consists of a pattern user's instantiation of a pattern template. After the user has selected their desired parameterisation, they can generate a framework instance. This process takes as input the framework template and a file containing the user's parameter settings. The user can then use the template viewer in the CO_2P_3S GUI to put their application-specific code into the hook methods of the framework, and generate a finished application.

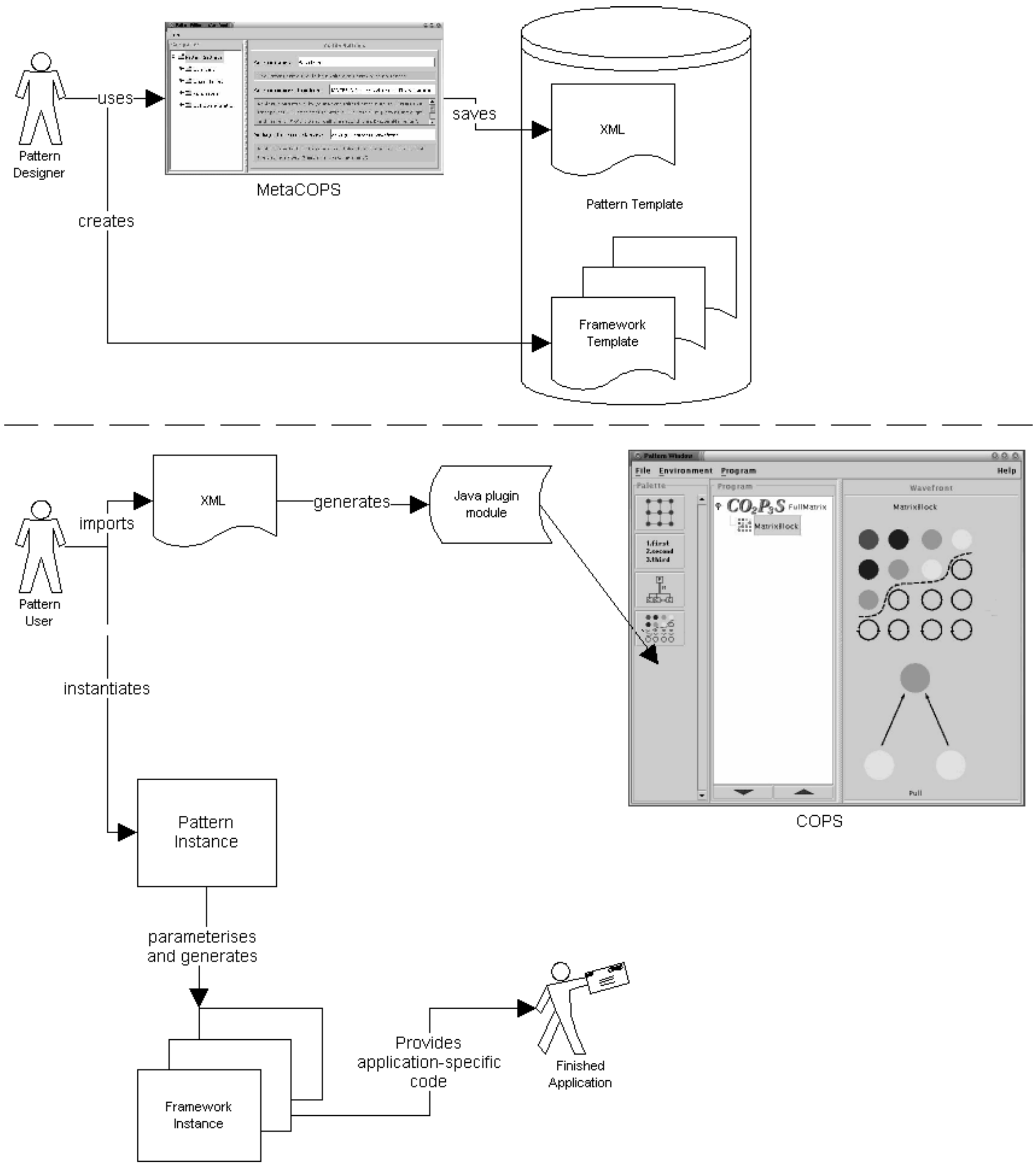


Figure 3.2: Files and interactions in CO_2P_3S and MetaCO_2P_3S.

3.4.1 XML for System-independent Pattern Template Storage

In addition to storing the pattern templates in a system-independent format, I chose to use a textual, human readable format, although this feature was secondary to expressiveness and machine readability. After some research we selected the XML[1] format [XML]. XML has become a well-known standard for data storage and sharing, and is published by the World Wide Web Consortium [W3C]. XML descended from SGML, and is similar to the HTML language used in web pages, but much more general. One important advantage of XML is that there a number of companion specifications published by the World Wide Web Consortium that greatly expand its usefulness. Furthermore, there is a large number of tools, including parsers, available for every computer platform.

XML is completely textual, and therefore serialises easily for storage on disk or transfer across a network. DTD[2] files (or the recent XML Schema standard) can be used to specify the allowable contents for a set of XML files. Since the requirements for pattern templates are strict and well-formed, I decided to have CO_2P_3S use a DTD to verify the format of XML files before they are imported into the programming environment. Appendix B describes the XML and DTD file formats used for MetaCO_2P_3S, and provides some examples.

XML is stored in a strict, hierarchical format. This fits well with the hierarchy of data required to create a pattern template. In fact, the MetaCO_2P_3S GUI mirrors this format by presenting a tree of choices to the pattern designer, each branch of which must be completed.

One specification related to XML that was particularly useful for my research was XSL[3] [XSL]. The intended purpose of XSL is to perform conversions on XML files. The documented uses of XSL included converting XML to HTML, mining the data to format it for different uses, or converting it to a binary format. XSL resembles a simple programming language, as it provides conditional tests, variables and operations for looping through lists.

I built a custom XSL style-sheet that converts the pattern template XML documents into Java source code files that subclass the PatternPane abstract class. This procedure, followed by a source file compilation, is performed automatically by the CO_2P_3S import feature, and requires no user intervention.

Since MetaCO_2P_3S saves the information entered by the pattern designer into XML transparently, and CO_2P_3S automatically converts it into a plug-in module using XSL, neither pattern users nor designers ever need to look at the pattern template XML files. However, this does not detract from my decision to use a human readable format, as it allows for easier pattern template debugging.

[1] Extensible Markup Language
[2] Document Type Definition
[3] Extensible Stylesheet Language

3.4.2 Javadoc for Framework Template Code Generation

Javadoc [Jav] is a tool, included with the Java distribution, whose original purpose was to generate HTML API documentation for Java libraries. Javadoc runs a modified Java compiler on Java source code files to parse the declarations and specially formatted comments. Javadoc comments have the following format:

```
/**
 * A comment describing the following Java construct.
 *
 * @sampleTag  a tag that is parsed by Javadoc
 */
public void sampleJavaDeclaration()
```

The most important features of Javadoc comment formatting are:

- the comment blocks must start with "/**".
- the comment blocks must end with "*/".
- the Javadoc tag names may either be predefined (i.e. the ones used by Javadoc to create API documentation) or user-defined.
- Javadoc comment blocks must immediately precede one of the following Java constructs:
  - a class declaration.
  - a constructor declaration.
  - a method declaration.
  - a field declaration.

Javadoc was eventually extended to allow pluggable *Doclets*. Doclets are Java programs that satisfy a contract allowing them to receive the parsed data from a Javadoc execution. This data includes the declarations and comments from each of the parsed classes. Method bodies and field initialisations are not provided, since they are ignored by Javadoc. The parsed Javadoc output is provided to the Doclet using the Doclet API [Doc], which provides access to the following information for each class:

- the imported classes and packages.
- the package of the class.
- the class declaration.
- the constructor declarations.
- the method declarations.

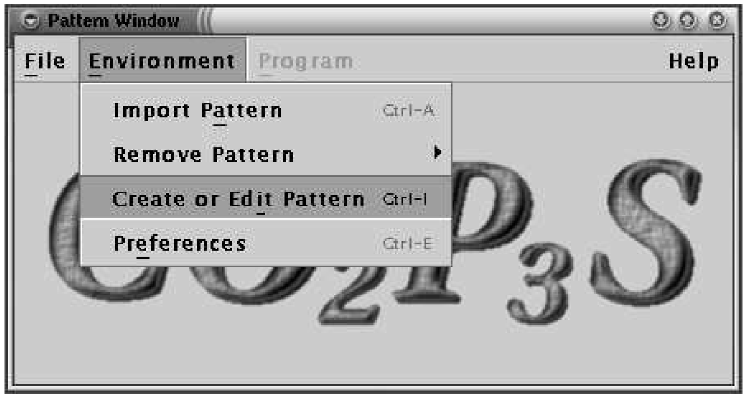


Figure 3.3: Launching the MetaCO_2P_3S editor.

- the field declarations.

For each of the declarations, Javadoc provides the text and tags from the associated comment, if any.

When I started to look at the code generation problem, and the replacement of the macro language used in the original CO_2P_3S implementation, my research eventually uncovered one group's use of Javadoc for code generation [Pol00]. Their experience only covered very simple uses, but I felt that it would be an ideal solution if extended. Javadoc allows us to have framework templates written in normal Java code, with the meta-programming hidden entirely in the comments. This means not only that my framework templates are much easier to read and edit, but also that they can be successfully compiled with no preprocessing, to aid in testing.

The new framework code generation implementation is a source code to source code transformation using Javadoc. There are two inputs to the process. One is a set of Java source code files that have been annotated by the pattern designer. The other is the pattern template parameters selected by a CO_2P_3S user.

The biggest pitfall that I experienced during my adoption of Javadoc as a framework code generator was its inability to provide method bodies through the Doclet API. My original intent was to modify the Javadoc parser to fix this drawback, but that proved impossible since the Javadoc parser source code was not available. To work around this short-coming, I decided to store the text for method bodies in separate files. The benefit to this was that a similar approach was already needed to provide parameterisation.

3.5 Pattern Template Creation using MetaCO_2P_3S

The MetaCO_2P_3S tool is launched from the CO_2P_3S GUI, as shown in Figure 3.3. The MetaCO_2P_3S GUI is pictured in Figure 3.4. This section describes the MetaCO_2P_3S portion of the pattern template creation process, using the Mesh as

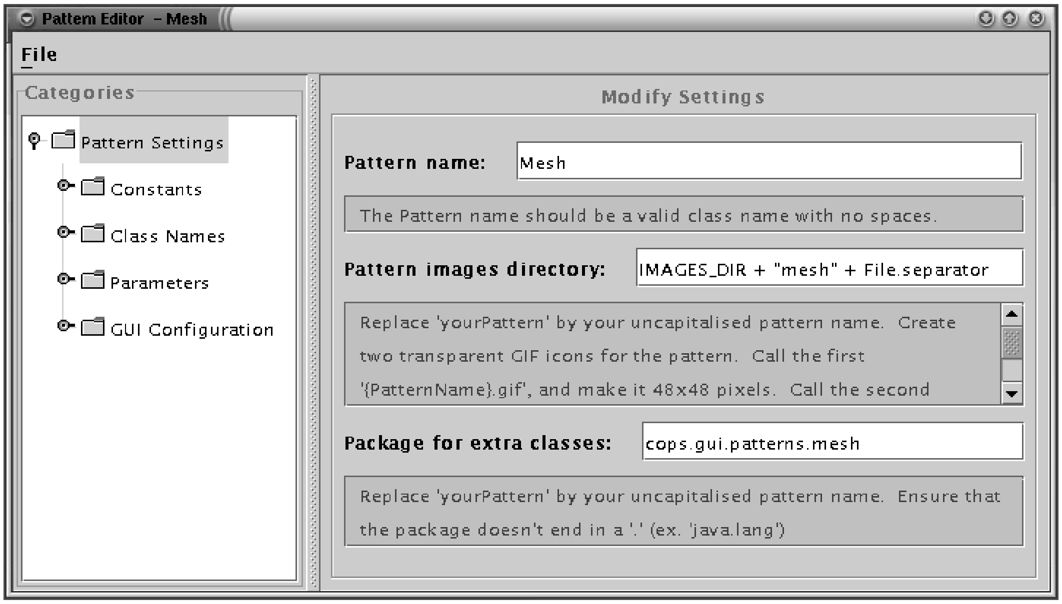


Figure 3.4: The Mesh Pattern Settings in MetaCO$_2$P$_3$S.

an example. Another example of the steps used in MetaCO$_2$P$_3$S during the pattern template creation process is given in Chapter 4.

In MetaCO$_2$P$_3$S, pattern templates can be saved to disk at any time, and loaded later for subsequent modifications. Figure 3.4 shows the Pattern Settings, the top level of the pattern template hierarchy, after the Mesh has been loaded into MetaCO$_2$P$_3$S. As this and subsequent figures show, MetaCO$_2$P$_3$S is well documented, informing the pattern designer about each field they need to fill in. This inline help serves to expedite the pattern creation process.

Figure 3.4 shows the two main panes in the MetaCO$_2$P$_3$S window. On the left is an expandable tree that represents the hierarchy of data in my definition of pattern templates, and also mirrors the layout of my XML files. Since a pattern template can have an arbitrary number of some elements such as class names and parameters, the pattern designer adds new leaf elements to the category tree using the buttons shown in Figures 3.6, 3.9 and 3.12. In the right pane, the pattern designer is responsible for filling in the necessary data fields, as directed by the inline help.

3.5.1 Pattern Settings in MetaCO$_2$P$_3$S

The settings pane for the Pattern Settings category serves to gather the pattern template identification details. It is here where the pattern designer enters the pattern name, and where the path names for extra files are defined. The inline help describes how to name the pattern icons, and where they should be located.

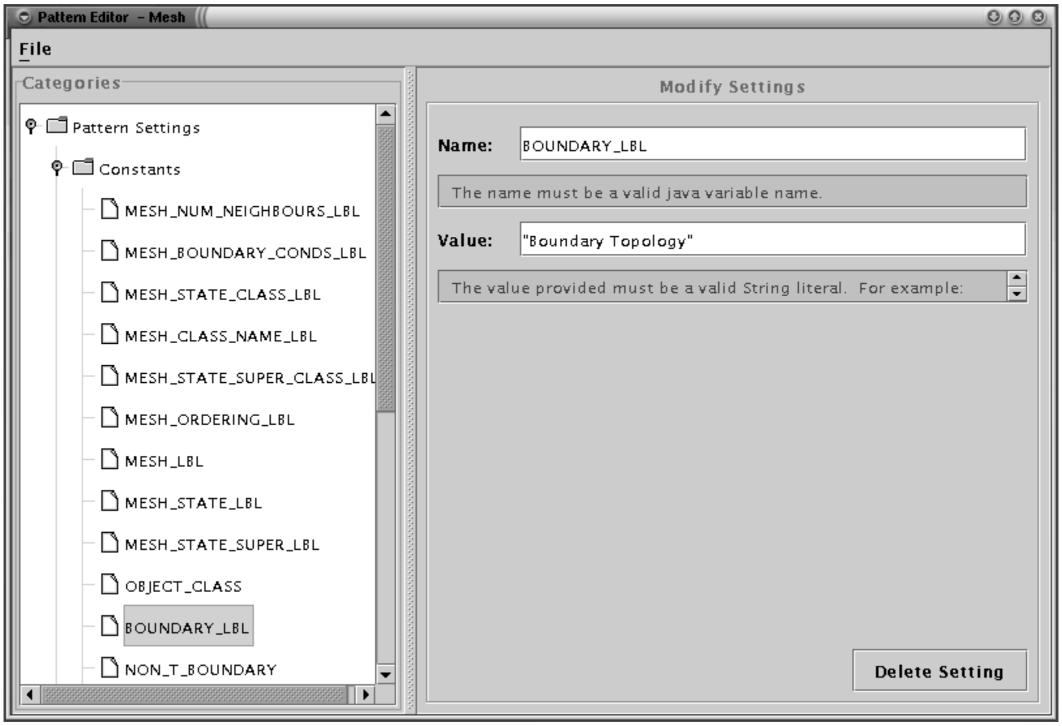


Figure 3.5: Constants used in the Mesh pattern.

3.5.2 Constants in MetaCO$_2$P$_3$S

The only category in the tree that was not mentioned in Section 3.1 is the one named Constants. Constants are string variables that the pattern designer can define and use in other fields, later in the pattern creation process. Figure 3.5 shows one such constant in the Mesh pattern template. Although I originally intended for the MetaCO$_2$P$_3$S GUI to provide constants in a selectable widget in all locations where their use is suitable, other implementation choices made this difficult, and it is not featured in the current version. Instead, when supplying information for fields needing a string literal, the pattern designer must either type a constant name (e.g. CONSTANT_NAME) or a string literal in quotation marks (e.g. "theString"). This nuance is described in the MetaCO$_2$P$_3$S inline help. The Delete Setting button at the bottom of the settings pane is a common feature in all leaf nodes, and allows the pattern designer to remove a setting that they no longer want.

3.5.3 Class Names in MetaCO$_2$P$_3$S

The Class Names category is shown in Figure 3.6. The pattern designer must specify every class that is to be involved in their framework template. I provide both *user-known* classes, shown in Figure 3.7 and *framework* classes, shown in Figure 3.8. Either type of class may be designated a *template class*, which is one that pattern users can enter application code into using hook methods. A pattern template can

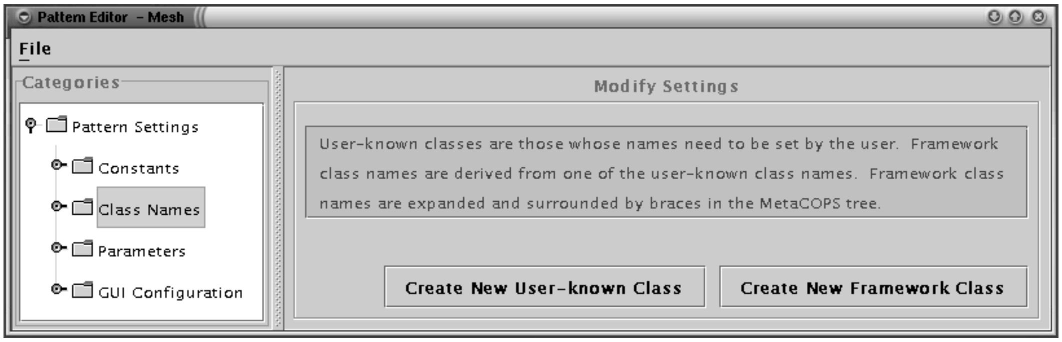


Figure 3.6: The Class Name pattern settings in MetaCO$_2$P$_3$S.

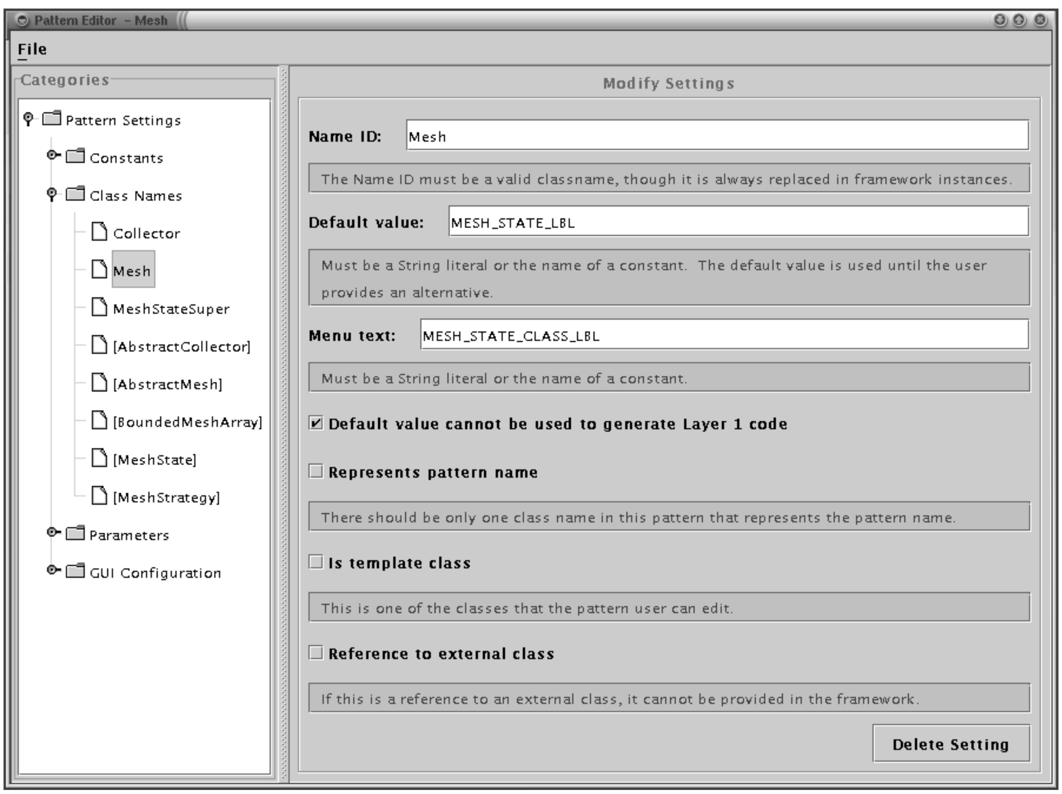


Figure 3.7: Editing a user-known class in the Mesh pattern template.

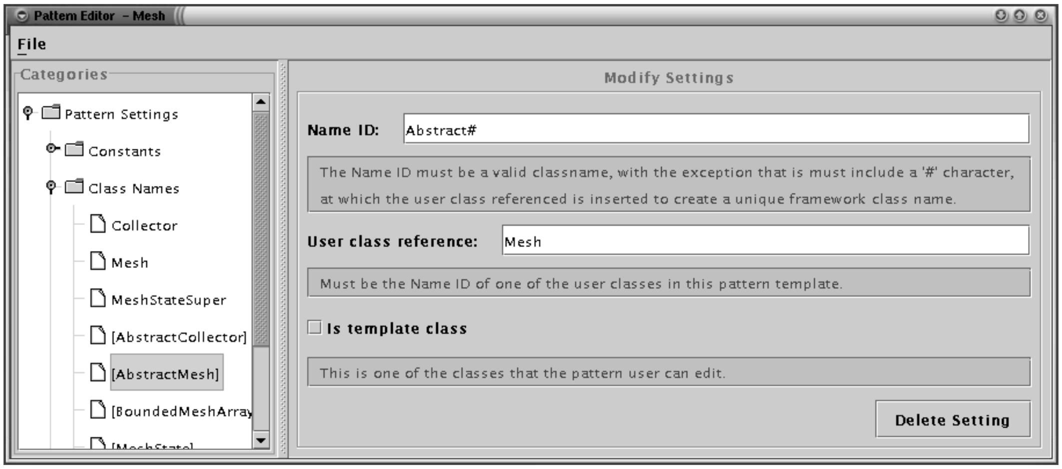


Figure 3.8: Editing a framework class in the Mesh pattern template.

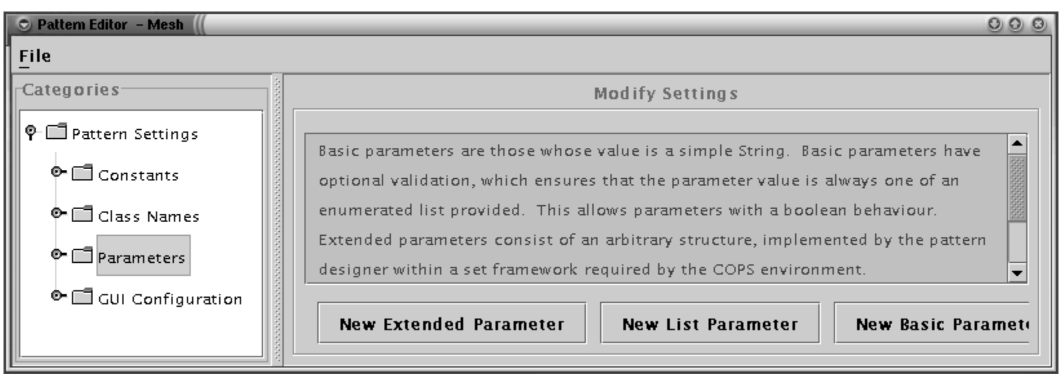


Figure 3.9: The Parameter pattern settings in MetaCO_2P_3S.

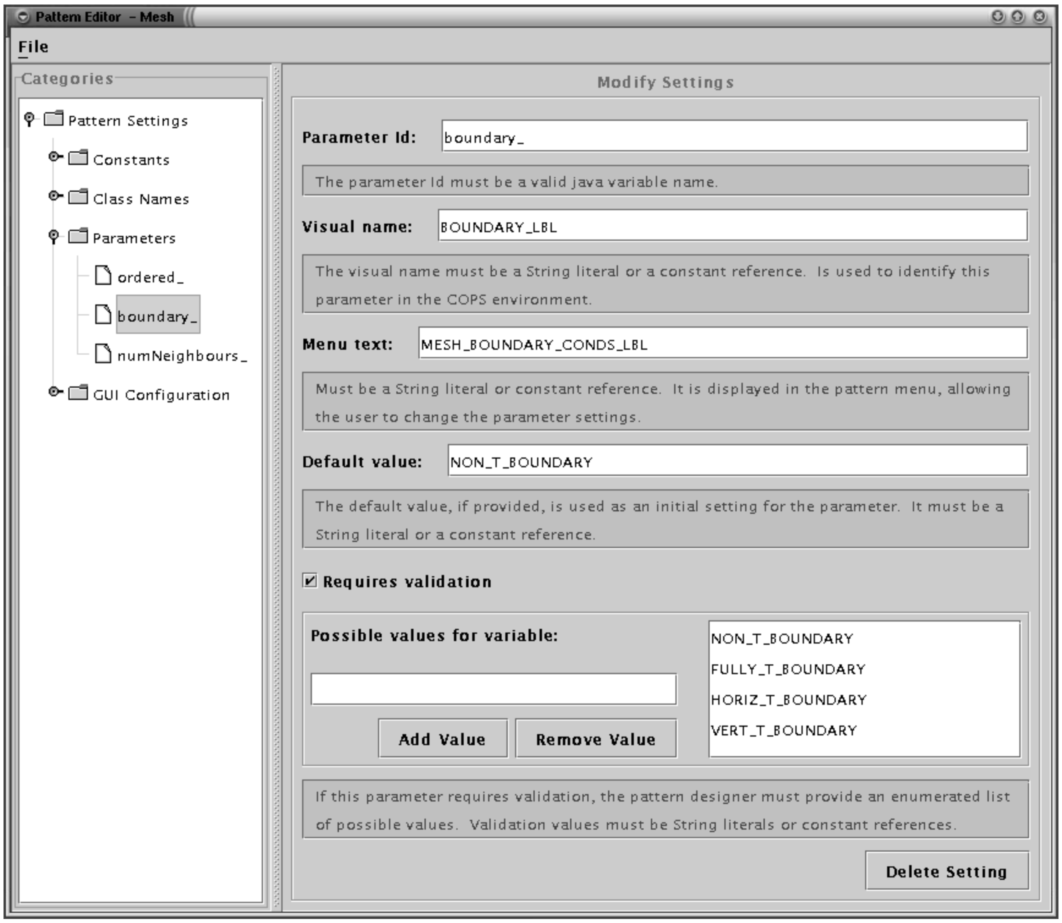


Figure 3.10: Editing a basic parameter in the Mesh pattern template.

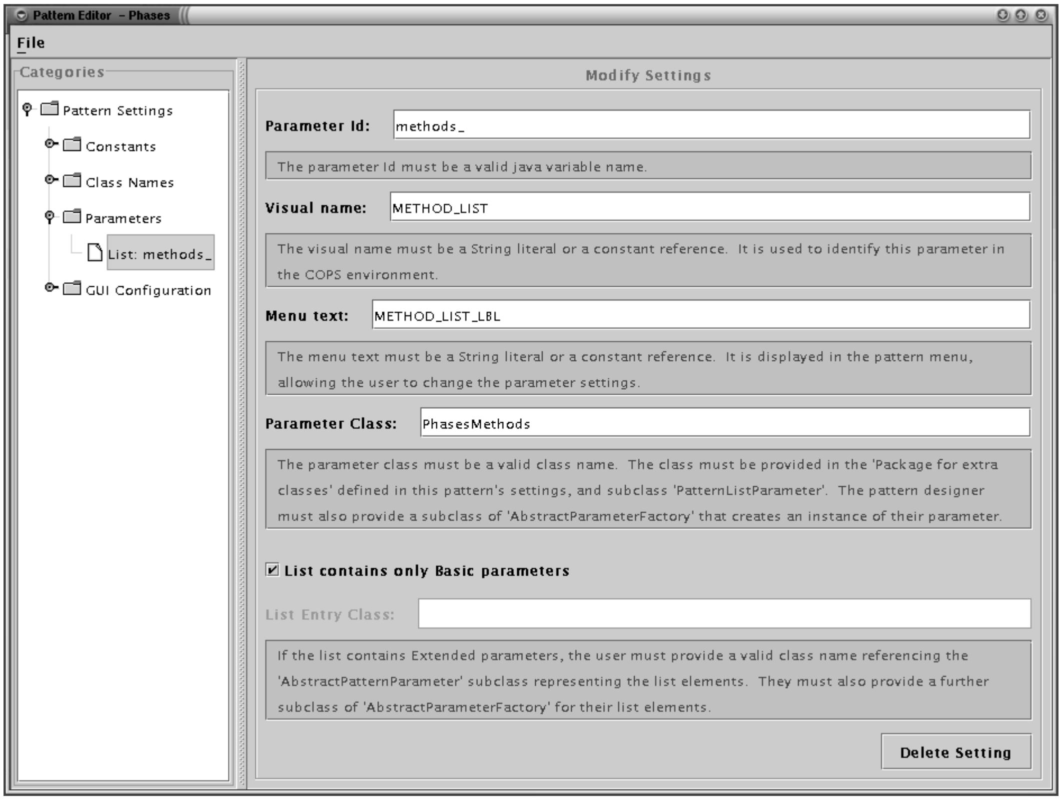


Figure 3.11: Editing a list parameter in the Phases pattern template.

have more than one *template class*.

User-known classes are those that are visible to pattern template users. One user-known class should be chosen by the pattern designer to represent the pattern instance in the CO_2P_3S GUI. The user-supplied name for this class will be displayed in the CO_2P_3S program panel. User-known classes can also be used to allow pattern users to integrate external classes into the framework, such as a superclass that provides application-specific functionality.

Framework classes are typically not known to the pattern user. To avoid name clashes, the pattern designer must add prefixes or suffixes to one of the user-known classes when forming a framework class name.

3.5.4 Parameters in MetaCO_2P_3S

The Parameters category is shown in Figure 3.9. The buttons that allow a pattern designer to add each of the three types of parameters defined in Section 3.1.3 are shown. Figures 3.10 and 3.11 show the settings pane for basic and list parameters, respectively. For each type of parameter, the pattern designer must provide a unique identification, and a name that will be displayed in the CO_2P_3S GUI.

The basic parameter has two additional pieces of required information. The first

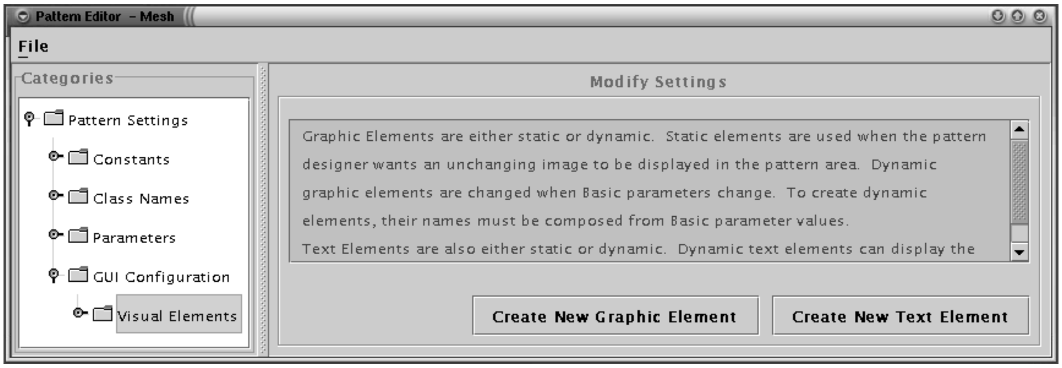


Figure 3.12: The visual GUI elements pattern settings in MetaCO$_2$P$_3$S.

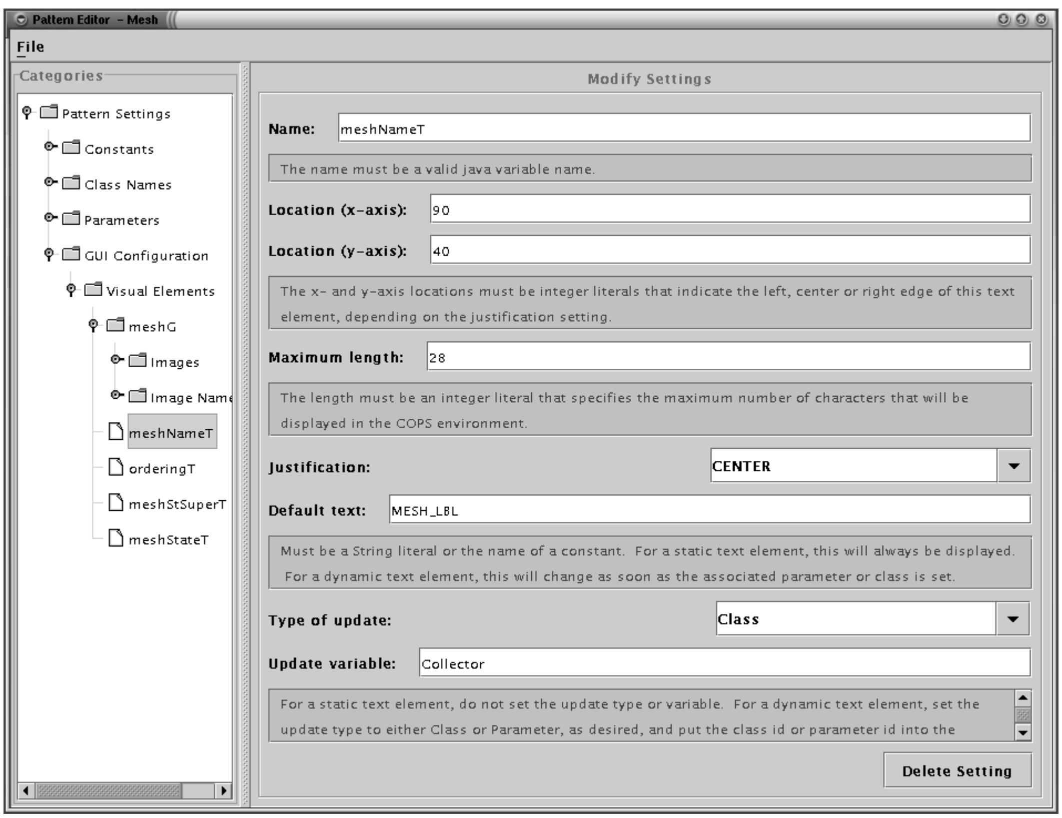


Figure 3.13: Editing a visual text element in the Mesh pattern template.

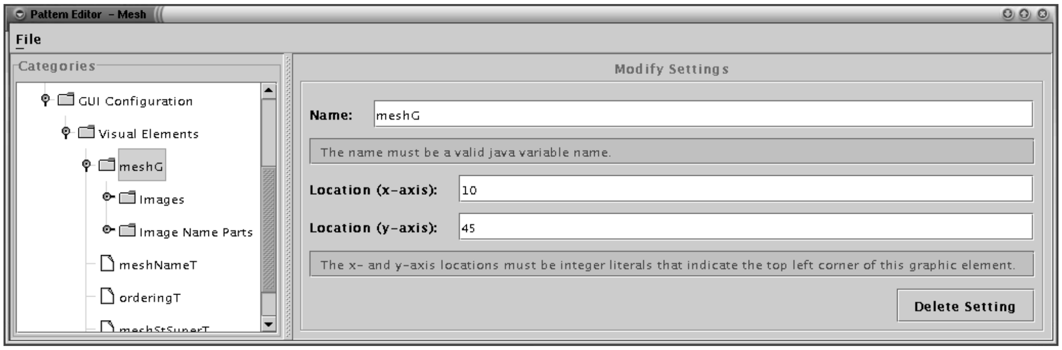


Figure 3.14: Editing a visual graphical element in the Mesh pattern template.

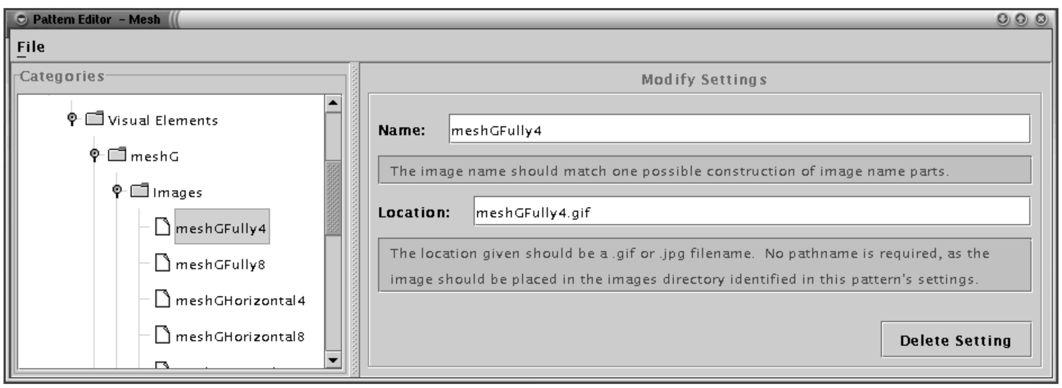


Figure 3.15: Adding images to a visual graphical element in the Mesh.

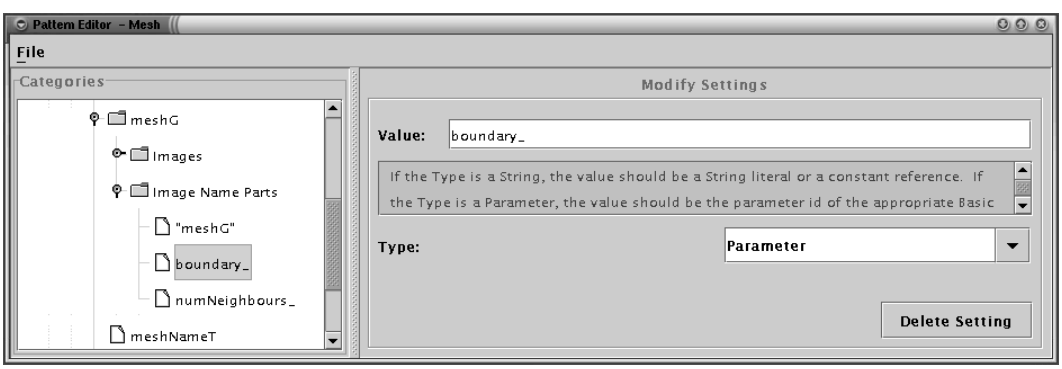


Figure 3.16: Configuring the name of a visual graphical element in the Mesh.

is an optional default parameter value. The second, for basic parameters with an enumeration of possible choices, is the list of valid options that are to be presented to the pattern user.

List parameters also have two additional pieces of required information. First is the parameter class, to be written by the pattern designer, that interfaces with the code generator to supply appropriate code to framework instances. Second, for list parameters with extended elements, is the parameter class for the extended parameters.

3.5.5 GUI Configuration in MetaCO$_2$P$_3$S

The GUI Configuration category is shown in Figure 3.12. The pattern designer can add both graphical and textual elements to the pattern template GUI display. Figure 3.13 shows the configuration settings for a text element. Figures 3.14 through 3.16 show the settings needed for graphical elements. Both element types require a name and a coordinate location for the pattern pane.

Text elements have a number of other configuration options. Two of these, the maximum length and justification deal only with the aesthetics of the visual display. Pattern designers must supply the default text string to be displayed. The text elements can also be set to update dynamically with the value of a given class name or parameter.

Graphical elements can also be made to dynamically represent the value of basic parameters. To achieve this, the pattern designer must specify one or more image name parts. Each of the parts can be a static string, or the value of a basic parameter. The concatenation of the parts is used to dynamically create a filename that should be found in the list of images supplied by the pattern designer. As exemplified by the Mesh pattern template, multiple parameter values can be used in the dynamic selection of a single graphical element.

3.6 Framework Template Creation

Given a pattern template and a specific parameterisation, CO$_2$P$_3$S must generate an appropriate object-oriented framework instance. The pattern designer must create a framework template that can accurately perform this task in conjunction with the MetaCO$_2$P$_3$S-generated pattern template description. It is the pattern designer's responsibility to ensure that the generated frameworks are error-free, and correctly implement the intended parallelism.

An annotated source code template must be written for each of the classes in the framework. It should be noted that I designed the framework template format to be used in a rich graphical editing environment. Because of this design, the annotations discussed in this section can easily be encapsulated in a programming environment. The following transformations must be supported by the supplied annotations:

- Placeholder class names in the annotated source files must be replaced with the unique names that are supplied by the CO_2P_3S user.
- Methods or variables may be selectively generated based on the user's basic parameter settings. The pattern designer must specify the combination of parameter settings that allows a given construct to be generated.
- Portions of method bodies may be selectively generated based on the given basic parameter settings.
- New methods or sections of method bodies may be generated based on extended or list parameter settings.
- In the template classes that were specified in MetaCO_2P_3S, selected methods must be marked as modifiable by the user. This allows CO_2P_3S to display hyper-links in the code template viewer.

The pattern description file created by MetaCO_2P_3S is stored within the CO_2P_3S installation directory as "`patterns/[PatternName].xml`". When the pattern template is saved, MetaCO_2P_3S also creates the framework template directory, at "`patterns/[PatternName]/`". The directory layout for patterns is depicted at the end of this section, in Figure 3.23. The framework template directory has the following contents:

- Java source files for each of the class names provided in MetaCO_2P_3S. The source files each have the prefix "`FrameworkCLASS_`", to clearly differentiate them from classes in framework instances. MetaCO_2P_3S automatically creates these files, which must then be edited by the pattern designer as discussed in Section 3.6.1.
- a "`framework_methods`" directory that contains subdirectories for each of the classes in the pattern. These subdirectories are where the pattern designer must put the default method bodies for each of their classes. The subdirectories are automatically created by MetaCO_2P_3S.
- a "`framework.prop`" Java properties file that links extended and list parameters (if any) to the Factory class used to create them. This file is created automatically by MetaCO_2P_3S.

3.6.1 Building the Framework Classes

The framework classes need to be written as compilable Java source files. Since the framework code generator uses the Javadoc tool, Javadoc comment formatting features prominently in these classes. They are written with all the necessary method declarations, but no method bodies. Default method bodies are put into separate files. The code generator does not support inner classes, although interfaces and abstract classes are supported. However, it is recommended that pattern designers supply dummy method bodies as needed, to ensure that the template can be compiled.

The Template Classes

The template classes are those that were selected by the pattern designer to be visible to the user. In the CO_2P_3S Template Viewer, the pattern user does not have the ability to edit method signatures in these classes. Instead, they are shown an unmodifiable view of the entire class, and allowed to click on hyper-links for the hook methods specified by the pattern designer.

To make hook methods, the pattern designer needs to put an "`@editable`" tag in the preceding Javadoc comment. To allow the user to add their own methods to the template class, the "`@userCodeAllowed`" tag should be put in the Javadoc comment preceding the class declaration. If the pattern designer wishes to allow the user to import additional classes or packages to the template class, they can place an "`@userImports`" tag in the class declaration Javadoc comment.

Figure 3.17 displays an excerpt from the MeshState template class from the Mesh, including some of the Javadoc tags described above. Figure 3.18 shows the output from CO_2P_3S after a pattern user has selected one possible parameterisation of the Mesh. Given this parameterisation as input, the code generator creates the Java framework instance code shown in Figure 3.19, and provides the CO_2P_3S GUI with the template class shown in Figure 3.20. The following sections provide further descriptions of the inputs and outputs to the code generation example shown in these figures.

The Connection to MetaCO_2P_3S

The framework template example in Figure 3.17 necessarily has connections to the information given by the pattern designer in MetaCO_2P_3S. Each of the class names or types that are prepended by "`FrameworkCLASS_`" are placeholder references to framework classes, and will be replaced with user-supplied names during framework instance generation. In addition, each instance of the "`@parameter`" Javadoc tag is directly followed first by the name of a parameter, as supplied by the pattern designer in MetaCO_2P_3S, and second by one of the enumerated values for that parameter.

```
/*****************************************************************************
 * This class represents a single node in a mesh computation.
 *
 * @userImports
 * @userCodeAllowed
 * @frameworkSuperclass FrameworkCLASS_MeshStateSuper
 */
public class FrameworkCLASS_MeshState
{

    /**
     * @initialValue 50
     */
    public static final int MAX_COUNT;

    /**
     * Constructor.
     *
     * @editable
     */
    public FrameworkCLASS_MeshState( int i, int j, int surfaceWidth,
        int surfaceHeight, Object initializer )
    {
    }

    /**
     * Iteration method for a top right corner node in an 8 point mesh.
     *
     * @parameter numNeighbours_ 8
     * @parameter boundary_ Non
     * @editable
     */
    public void topRightCorner( FrameworkCLASS_MeshState south,
        FrameworkCLASS_MeshState southwest, FrameworkCLASS_MeshState west )
    {
    }

    /**
     * Iteration method for a top right corner node in a 4 point mesh.
     *
     * @parameter numNeighbours_ 4
     * @parameter boundary_ Non
     * @editable
     */
    public void topRightCorner( FrameworkCLASS_MeshState left,
        FrameworkCLASS_MeshState down )
    {
    }
}
```

Figure 3.17: Mesh template code example.

```
#MeshClass User Properties
PatternName=MeshClass
ordered_=Ordered Computation
boundary_=Non
numNeighbours_=4
FrameworkCLASS_MeshStateSuper=Object
FrameworkCLASS_BoundedMeshArray=BoundedMeshNodeArray
FrameworkCLASS_MeshState=MeshNodeState
FrameworkCLASS_Mesh=MeshNode
```

Figure 3.18: User parameterisation from CO_2P_3S.

The Java construct immediately following the Javadoc comments containing these tags are generated only if the pattern user selects the given parameterisation in CO_2P_3S prior to framework generation.

External Class References

The pattern designer may use the "`@frameworkSuperclass [referenceClass]`" tag in the Javadoc comment directly preceding the class declaration if they wish to allow pattern users to supply an external superclass for one of the framework classes. The supplied `referenceClass` should be one of the class identifiers supplied in MetaCO_2P_3S for which the "Reference to external class" check box is selected. Figure 3.17 shows an example usage of this Javadoc tag. The generated framework code shown in Figure 3.19 does not have a superclass, since the user-supplied parameterisation of Figure 3.18 does not include a setting for the appropriate `referenceClass` (although it has the default Java superclass setting of Object).

Initialising Class Fields

If the pattern designer wants to declare class-level fields, such as instance variables or constants, they can simply use normal Java syntax. However, if a field initialisation is required, the pattern designer must attach a Javadoc comment with a "`@initialValue [value]`" tag. The initialiser should not include either an "`=`" sign or a "`;`" terminator. An example initialisation is shown in Figure 3.17, for the MAX_COUNT constant.

Conditional Construct Generation Using Basic Parameters

The pattern designer can specify which methods, constructors and fields should be generated by using the "`@parameter [paramId] [paramValue]`" tag in the preceding Javadoc comment. The `paramId` refers to the parameter identification from MetaCO_2P_3S. The `paramValue` refers to the user-selected value of the given parameter.

```
// user imports

/*****************************************************************************
 * This class represents a single node in a mesh computation.
 */
public class MeshNodeState
{

    /**
     * Constructor.
     */
    public MeshNodeState( int i, int j, int surfaceWidth, int surfaceHeight,
                          Object initializer )
    {
    }

    /**
     * Iteration method for a top right corner node in a 4 point mesh.
     */
    public void topRightCorner( MeshNodeState left, MeshNodeState down )
    {
    }

    // user code

    public static final int MAX_COUNT = 50;
}
```

Figure 3.19: Mesh template code in framework instance after parameterisation.

If multiple entries are supplied with the same parameter identification (but different values), generation of the associated construct occurs if any one of the given settings is true. This gives the ability to selectively generate constructs using Boolean "OR" logic. If multiple entries are listed, each with different parameter identifications, the associated Java construct is generated only if all of the given values are true. This Boolean "AND" logic can be combined with the "ORing" of values on a single parameter identification to create expressive and powerful conditional code generation statements.

Figure 3.17 gives an example of conditional construct generation using basic parameters. Since the conditions in the second iteration method header match the user-supplied parameterisation shown in Figure 3.18, only the second method gets generated to the framework code shown in Figure 3.19.

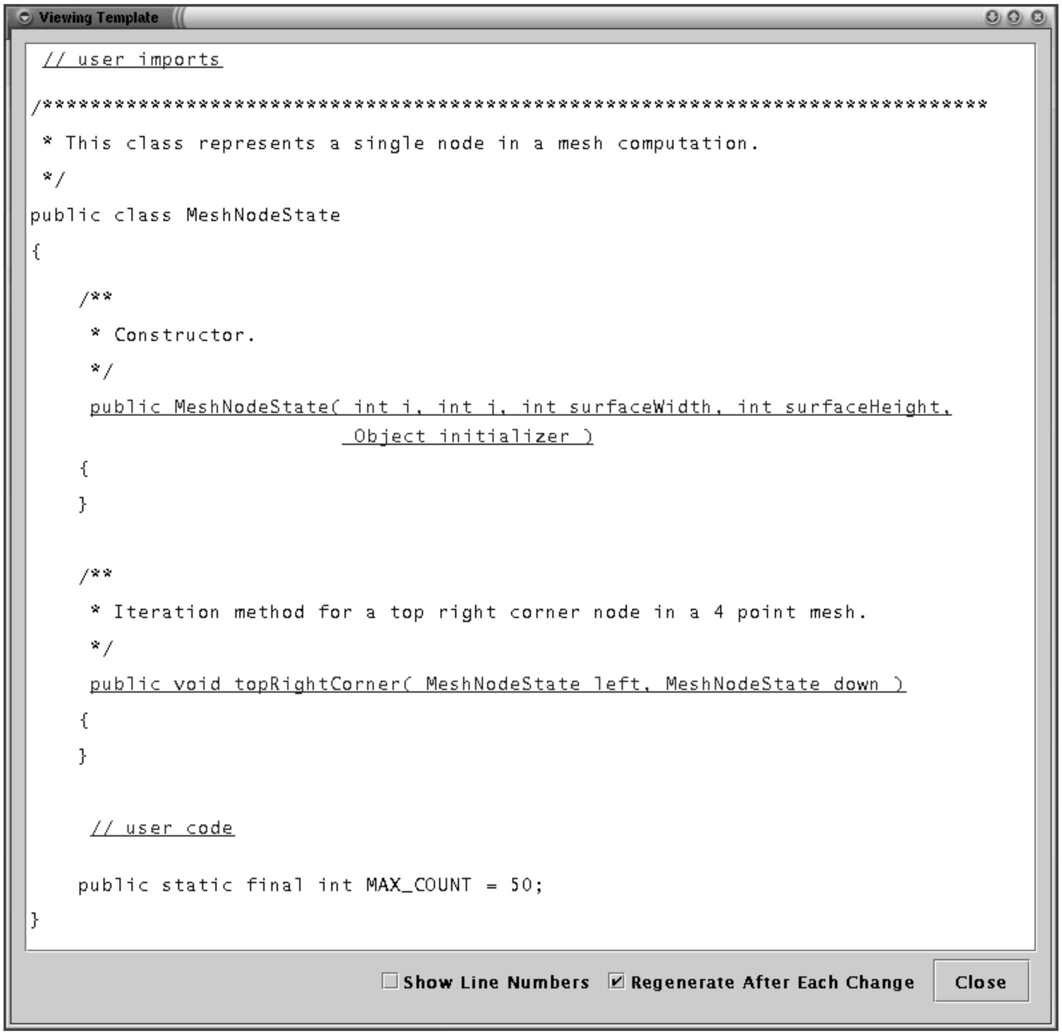


Figure 3.20: Mesh template code in CO_2P_3S Template Viewer after parameterisation.

```
// This method body is located at:
// patterns/Mesh/framework_methods/FrameworkCLASS_AbstractMesh/topRightCorner


// The following class name, enclosed in # characters, gets replaced with
// the user class name in the pattern template instance at framework code
// generation time
#FrameworkCLASS_BoundedMeshArray# state = state_;

// The following text within the MACRO delimiters is only generated when the
// pattern user sets the number of neighbours to eight
#FrameworkMACRO#(numNeighbours_ == 8)
<iteration code deleted>
executeInteriorNodes(state, 0, width - 1, 1, height) ;
#FrameworkMACROend#

#FrameworkMACRO#(numNeighbours_ == 4)
<iteration code deleted>
executeInteriorNodes(state, 0, width - 1, 1, height) ;
#FrameworkMACROend#
```

Figure 3.21: Mesh method body framework template example.

```
BoundedMeshNodeArray state = state_;

<iteration code deleted>
executeInteriorNodes(state, 0, width - 1, 1, height) ;
```

Figure 3.22: Mesh method body generated after parameterisation.

Construct Generation Using Extended and List Parameters

The pattern designer can add code to a class from extended or list parameters by putting the "`@extParameter [paramId]`" tag in the Javadoc comment preceding the class declaration. Further discussion of extended and list parameter code generation is in Section 3.6.2.

Supplying Default Method Bodies

Each default method body needs to be supplied in a separate file in the subdirectory of framework_methods matching the containing framework class. For methods marked `@editable`, the given default method bodies are used until they are modified by the pattern user. The filenames for default method bodies must be unique within a class, so they consist of the method or constructor name followed by the each of the parameter types. Each of the name is separated by a "." character. For example, a method with the signature "`public void read( Reader r, boolean b );`"

would have its default method body stored in a file called "`read.Reader.boolean`."

Method Body Expansion

Method bodies may contain macros that cause their contents to vary based on a given parameter value. The framework code generator expands these macros using the following order of operations:

1. sections of code wrapped by basic parameter macros are generated only if the given parameter conditions hold,

2. code fragments are inserted using extended or list parameter macros, and

3. placeholder class names are replaced by user-supplied class names.

Macros for basic parameters must have delimiters at the beginning and end of the code fragment that is to be conditionally generated. The syntax of the opening delimiter is "`#FrameworkMACRO#([paramId] [op] [value])`," where `paramId` refers to the parameter identification, `op` can be either "`==`" or "`!=`" to indicate the conditional test to be performed, and `value` is the value to test in the conditional statement. The syntax of the closing delimiter is simply "`#FrameworkMACROend#`." The code fragment contained within the macro delimiters is generated only if the given conditional statement is true. The macros can be nested, so that code fragments will be conditionally generated based on multiple parameter values.

The syntax of macros that insert code fragments based on extended and list parameter settings is "`#FrameworkPARAM_[paramId]#`." The `paramId` is the appropriate parameter identification. The expansion of these macros is discussed further in Section 3.6.2.

Since placeholder class names need to be replaced by user-supplied class names during code generation, their references in default method bodies must be specially encoded. The encoding takes the form "`#FrameworkCLASS_[className]#`," where `className` refers to the class name identification supplied in MetaCO_2P_3S.

Given the user-supplied parameterisation from Figure 3.18, the default method body example in Figure 3.21 generates to the framework code in Figure 3.22.

3.6.2 Implementing Extended and List Parameters

The Distributor and Phases pattern templates both use list parameters. The implementations of these pattern templates serve as an excellent example for pattern designers that need to use either extended or list parameters. The first part of the extended parameter design requires that a parameter class provided with CO_2P_3S be sub-classed to provide a module that can be used to obtain and store parameter settings in CO_2P_3S. Since the code generator module runs in a sand box that is decoupled from CO_2P_3S, the second piece of the design is a class that can create an instance of the appropriate parameter. This class must subclass a provided template that is designed using the well known Abstract Factory design pattern [GHJV95].

During code generation, extended and list parameters can be used either to add code at the class level, such as methods or fields, or to add code fragments to method bodies. Both cases are treated similarly. Since extended parameters can store information in an arbitrary format, the pattern designer must write code to define what gets generated. This code must be placed in the package defined in the Pattern Settings of MetaCO_2P_3S, which is typically "`cops.gui.patterns.[patternName]`."

The pattern designer must provide a means by which the code generator can gain access to an extended parameter. This is done by creating a subclass of the "`AbstractParameterFactory`" class, which is included in the CO_2P_3S distribution. The name of the factory subclass is written by MetaCO_2P_3S into the "`framework.prop`" file. The only requirement of the factory class is that it implement a method that calls the constructor of the appropriate extended parameter class. The location of each of the files making up a pattern template is depicted in Figure 3.23.

The parameter class must also be written by the pattern designer. For extended parameters, the parameter class subclasses "`AbstractPatternParameter`." For list parameters, it subclasses "`PatternListParameter`." Each of these super-classes is provided with CO_2P_3S.

In `AbstractPatternParameter` subclasses, the pattern designer must implement methods that perform the following functions:

- test whether a parameter has been set by the pattern user, and is therefore ready for code generation,
- create a dialog to gather parameter settings from the user,
- save the parameter settings to a string,
- load the parameter settings from an equivalent string, and
- provide code to the framework generator for a given class.

For list parameters, CO_2P_3S automatically handles the first four functions.

To provide methods and fields to the code generator, the pattern designer must use the "`CopsMethod`," "`CopsField`" and "`CopsArgument`" data types supplied with the CO_2P_3S distribution. The purpose of these classes is to provide code fragments to the framework generator in a structured and standardised format. Method body code fragments are supplied to the code generator using simple strings.

Name	Type
copsProj	Directory
libs	Directory
cops	Directory
programs	Directory
patterns	Directory
Mesh	Directory
framework_methods	Directory
FrameworkCLASS_AbstractMesh	Directory
FrameworkCLASS_AbstractCollector	Directory
FrameworkCLASS_MeshStrategy	Directory
FrameworkCLASS_BoundedMeshArray	Directory
FrameworkCLASS_Mesh	Directory
FrameworkCLASS_Collector	Directory
FrameworkCLASS_MeshState	Directory
notDone	File
FrameworkCLASS_MeshState.java	File
FrameworkCLASS_AbstractMesh.java	File
FrameworkCLASS_BoundedMeshArray.java	File
FrameworkCLASS_AbstractCollector.java	File
FrameworkCLASS_Mesh.java	File
FrameworkCLASS_Collector.java	File
FrameworkCLASS_MeshStrategy.java	File
framework.prop	File
Wavefront	Directory
Distributor	Directory
Mesh.xml	File
Distributor.xml	File
Wavefront.xml	File
Phases.xml	File
Phases	Directory
binUtil	Directory
XSL	Directory
CopsPattern.xsl	File
DTD	Directory
CopsPattern.dtd	File
images	Directory
Mesh.gif	File
smallMesh.gif	File
Wavefront.gif	File
Distributor.gif	File
Phases.gif	File
smallWavefront.gif	File
smallDistributor.gif	File
smallPhases.gif	File
runCops	File

Figure 3.23: Directory layout of pattern templates in CO_2P_3S.

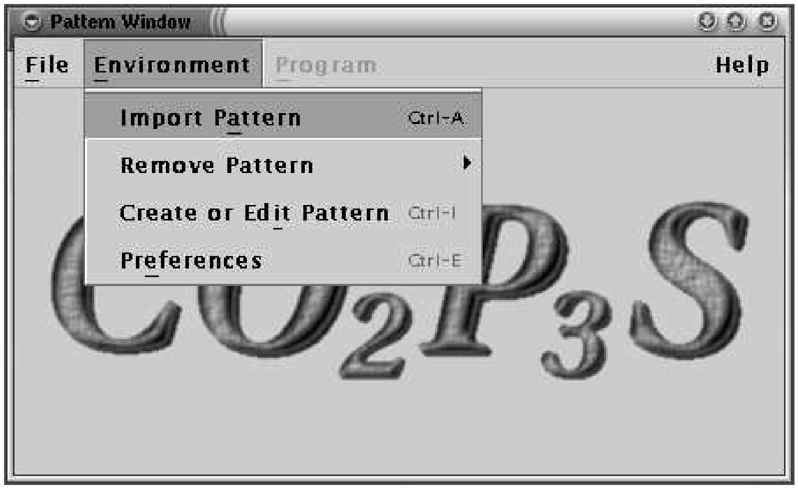


Figure 3.24: Importing a pattern template into CO_2P_3S.

3.7 Testing the Pattern Template

It is crucial that pattern designers thoroughly test their pattern template creations prior to their release. As part of this testing, the pattern templates can be imported into the CO_2P_3S environment, as described in the following section. If changes are made to the pattern template, the pattern designer can easily update the pattern template in CO_2P_3S for further testing.

3.8 Importing Pattern Templates into CO_2P_3S

I have made it easy to import new parallel design pattern templates into CO_2P_3S, or to update existing ones. The CO_2P_3S user simply needs to select the "Import Pattern" menu item from the "Environment" menu as shown in Figure 3.24, then browse to the appropriate directory and select the desired pattern template file. During the import process, CO_2P_3S automatically converts the selected XML file into a plug-in Java module, and adds a button containing the pattern icon to the palette.

Chapter 4

Validating MetaCO_2P_3S

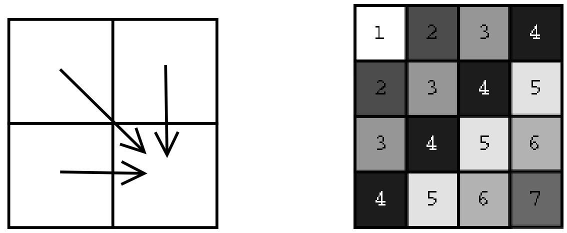
I designed and implemented the CO_2P_3S meta-programming extension so that it would allow any design pattern to be transformed into a pattern template. This section describes the steps I have taken to validate my approach.

4.1 Recreating CO_2P_3S

My first step in testing the coverage and correctness of the MetaCO_2P_3S tool was to regenerate each of the pattern templates from the original CO_2P_3S environment. The Mesh, Distributor and Phases pattern templates have all been successfully regenerated, and the standard CO_2P_3S distribution now is the one generated by MetaCO_2P_3S. Since these pattern templates formed a basis for the MetaCO_2P_3S development, their creation was not completely straightforward. However, as MetaCO_2P_3S was continuously refined to deal with the issues that arose during their creation, I could see that my work was greatly simplifying the overall pattern creation process. The short pattern template development times required for the work described in Sections 4.2 and 4.3 are evidence of my tool's success in attaining the goal of enabling and simplifying pattern creation.

4.2 Case Study: Genetic Sequence Alignment

A common problem in bioinformatics lies in finding an optimum alignment for a pair of DNA or protein sequences [CSS00]. Typical algorithms for sequence alignment construct a dynamic programming matrix with the sequences on the top and left edges. A score is propagated from the top left corner to the bottom right. The value of each entry in the matrix depends on three previously computed values, above, to the left, and in the above-left diagonal, as shown in Figure 4.1(a). Once all of the values in the matrix have been calculated, another algorithm can be used to trace backwards through the matrix to get the maximal cost path, or optimum sequence alignment.



(a) Score propagation.

(b) The wavefront computation ordering.

Figure 4.1: Solving the sequence alignment problem with a dynamic programming matrix.

4.2.1 Isolating the Wavefront Design Pattern

John Anvik, one of the graduate students in our research group, was studying the dynamic programming problem, and attempted to parallelise it using CO_2P_3S. He identified a wavefront parallel design pattern in the dynamic programming algorithm, but no Wavefront pattern template was available, preventing further progress.

Wavefront design patterns apply to problems where a computation needs to sweep breadth-first through a tree, with child nodes having data dependencies on their parents. The *wavefront* describes the edge separating the processed nodes at the top of the tree from the nodes waiting to be processed. The dynamic programming problem is easily expressed as a wavefront due to the dependency of each matrix entry on three of its neighbours. Figure 4.1(b) shows how the data dependencies in Figure 4.1(a) can be transformed to a wavefront computation. Blocks with the same number are computed concurrently after the blocks with smaller numbers have been computed. The wavefront design pattern can be implemented using a work queue, where nodes at the edge of the wavefront whose data dependencies have been satisfied are available to be computed. A user's view into a wavefront framework requires only that they provide the node processing implementation. A single parameter affects the implementation of the wavefront design pattern, determining whether notifications of computation completion are pushed to child nodes, or pulled from parents.

4.2.2 Creating the **Wavefront** Pattern Template

The lack of a Wavefront pattern template in CO_2P_3S provided an opportunity to use MetaCO_2P_3S. John Anvik had not been involved in the MetaCO_2P_3S research, which also made the exercise of adding a pattern template a test-bed for the usability of my tool.

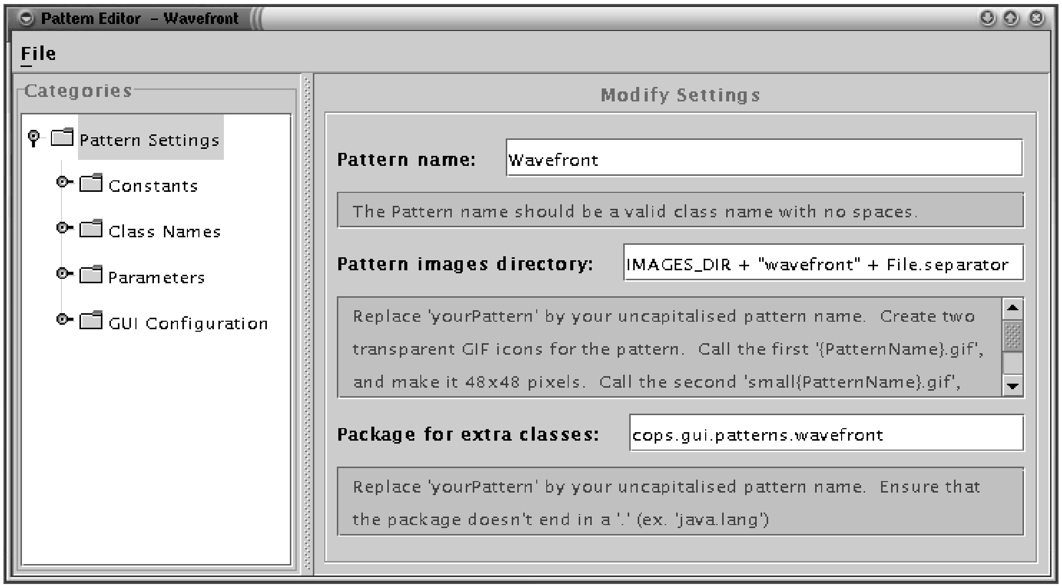


Figure 4.2: The Wavefront pattern template in MetaCO$_2$P$_3$S.

The first step used to create the Wavefront pattern template was to specify the pattern description using MetaCO$_2$P$_3$S. After launching the tool, John Anvik named the new pattern template and supplied an icon to identify the Wavefront in CO$_2$P$_3$S. Figure 4.2 illustrates this process. Note that the text, "yourPattern", supplied as part of the default pattern images directory has been replaced by "wavefront", as suggested by the accompanying inline help.

Next, the class names for the framework template were supplied. One of these, called *Wavefront*, was selected as a user-known class, and also as a user-modifiable class. Six framework classes were defined, the names of which were made dependent on *Wavefront* for their uniqueness, with suffixes added to indicate their role in the framework. The class settings in MetaCO$_2$P$_3$S are shown in Figure 4.3. Referring back to Figures 3.6 through 3.8, we see how Delete Setting buttons are at the bottom of every class (or parameter) setting, and that new classes (or parameters) are created using buttons at the bottom of their respective tree elements. Note that since the Default value and Menu text fields require a Java String literal, the quotation marks shown are required.

The lone Wavefront pattern parameter, described in Section 4.2.1, was defined as a basic parameter using MetaCO$_2$P$_3$S. The parameter was called *notification*, and was given an enumeration of two possible values: *push* and *pull*. Figure 4.4 shows the *notification* parameter being defined.

The last step in the pattern description process was providing a GUI configuration. The result of this configuration is shown in Figure 4.5. At the top, a textual element is displayed that automatically updates to display the user-supplied name

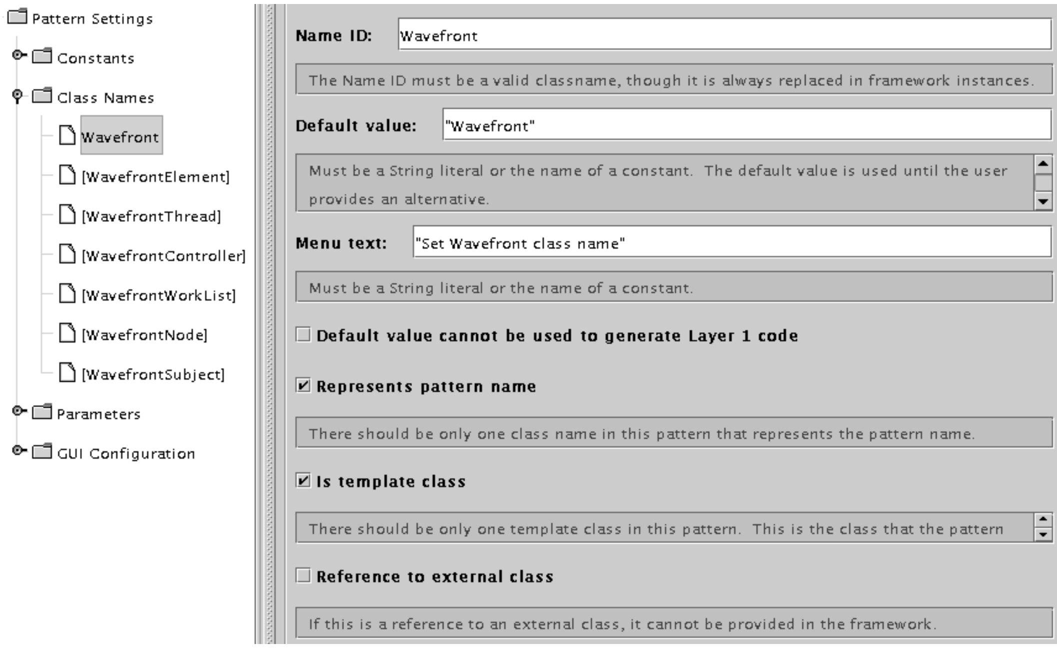


Figure 4.3: The Wavefront pattern template class settings.

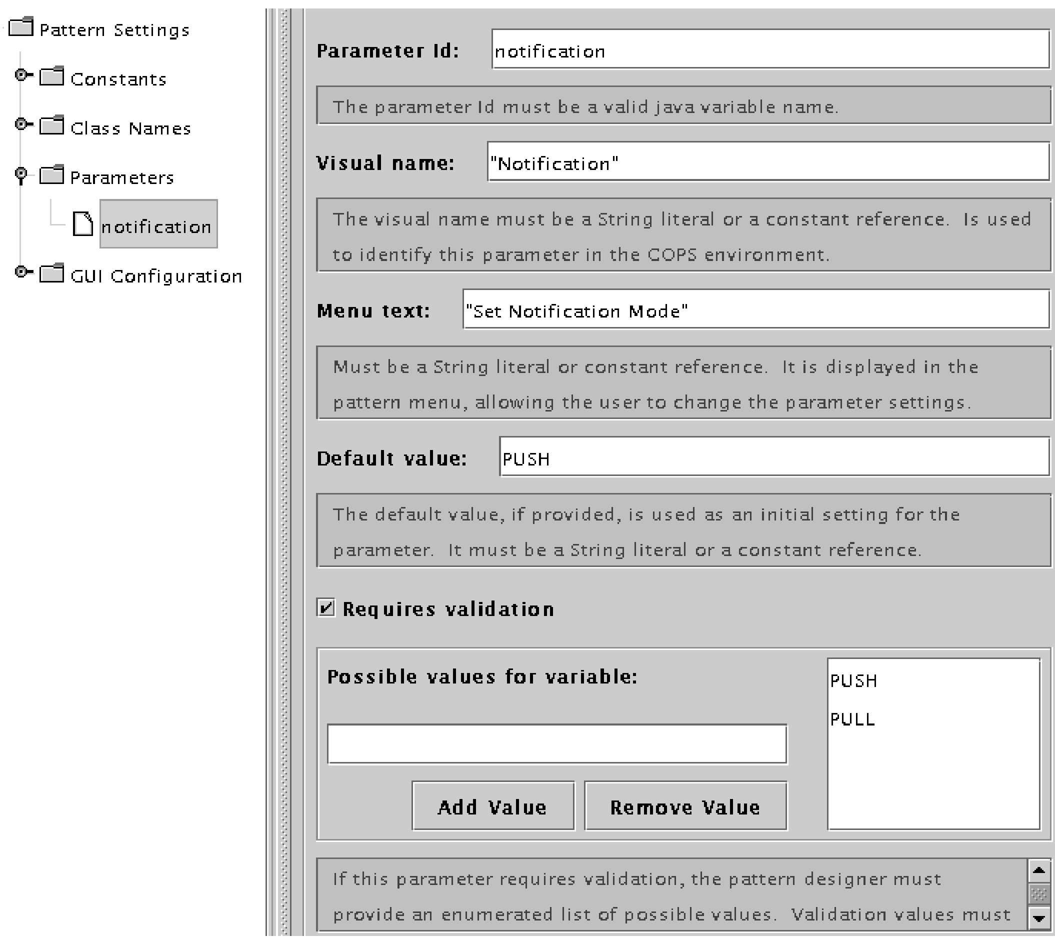


Figure 4.4: The Wavefront pattern template parameter settings.

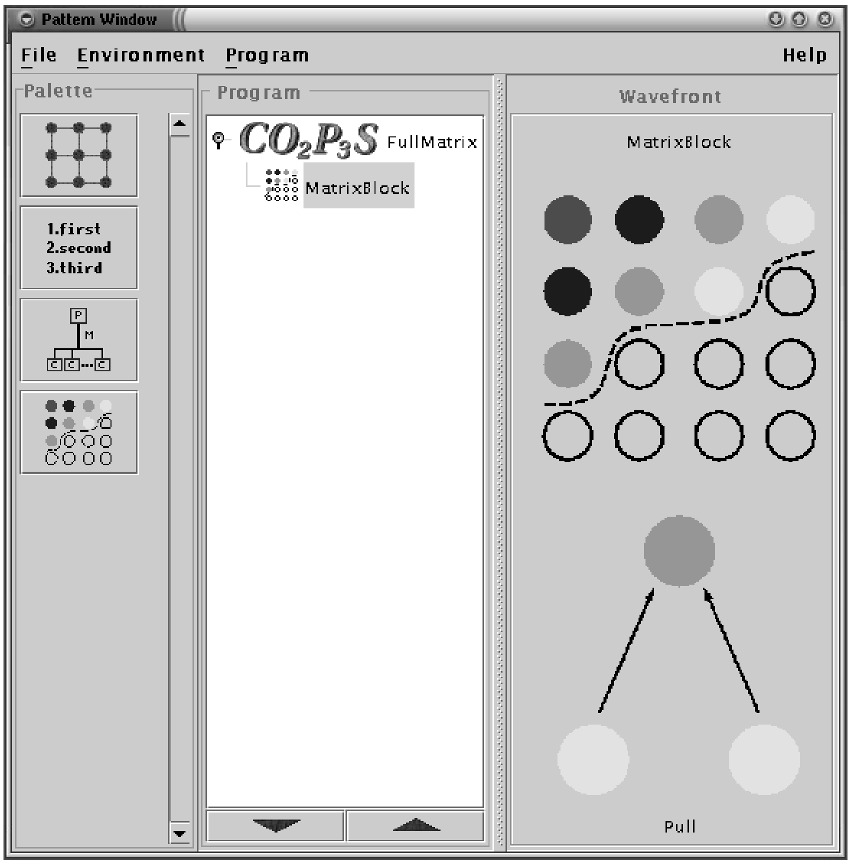


Figure 4.5: The Wavefront pattern template in CO_2P_3S.

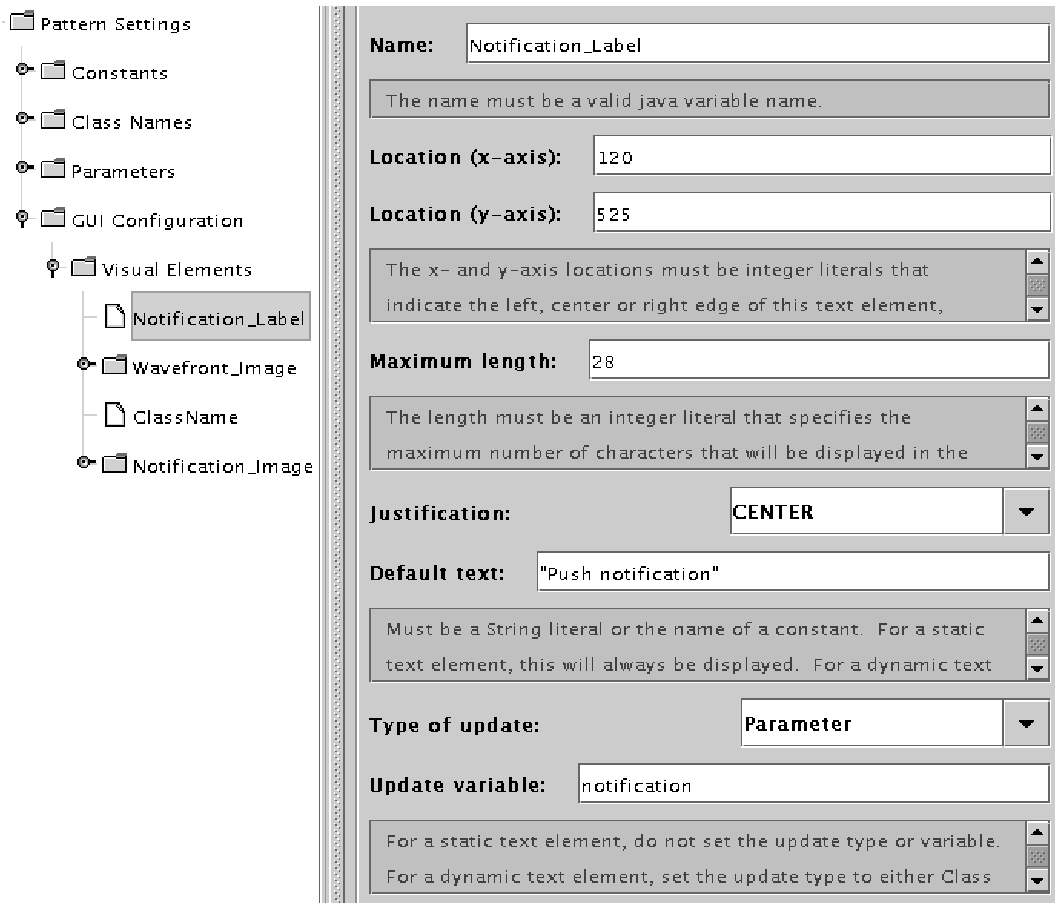


Figure 4.6: The Wavefront pattern template GUI settings.

Notification	Execution Time (seconds)			
	Seq	2P	3P	4P
Push	229.0	117.3	83.2	65.4
Pull	230.1	118.5	83.5	66.4

Table 4.1: Execution times using the Wavefront for sequence alignment.

for the *Wavefront* class. To do this, the pattern designer provided the text location, and the class name to display in $\mathrm{MetaCO_2P_3S}$. Below the class name is an image of a wavefront. The pattern designer provided this image and its location in the GUI using $\mathrm{MetaCO_2P_3S}$. At the very bottom of the display, the graphical figure and the text display are both representing the *notification* parameter setting. In $\mathrm{MetaCO_2P_3S}$ the pattern designer provided two images, and identified which one should be displayed with both possible parameter value. Figure 4.6 shows the textual representation of the *notification* parameter being defined in $\mathrm{MetaCO_2P_3S}$.

After entering the pattern description using $\mathrm{MetaCO_2P_3S}$, John Anvik needed to provide annotated framework source code for each of the defined classes. This entailed writing normal Java source code, with the addition of conditional compilation sections that depended on the setting of the *notification* parameter.

At this point, the initial version of the Wavefront pattern template was completely specified. John Anvik imported it into the $\mathrm{CO_2P_3S}$ environment, and tested the pattern template prior to implementing the sequence alignment dynamic programming program.

4.2.3 Analysis

The new Wavefront pattern template was used with $\mathrm{CO_2P_3S}$ to implement the dynamic programming matrix algorithm for genetic sequence alignment. Two sequences of 10,000 random proteins each were used as test data. The sequential and parallel implementations of the algorithm were run using a Java 1.3 virtual machine with native threads on a four-processor shared-memory SGI O2. The *push* and *pull* notification parameter settings were both used independently as a performance comparison. Table 4.1 shows the average execution times for 20 runs of each implementation. The parallel speedups are compared in Figure 4.7. There were no significant differences in the performance numbers for the *push* and *pull* notification parameter values. This could indicate that the parameter is unnecessary. If so, $\mathrm{MetaCO_2P_3S}$ makes it easy to remove the parameter from the pattern template.

The Wavefront pattern template described in this dissertation is still undergoing modifications. There is work being done in our research group to refine the pattern template with new parameters, making it more general.

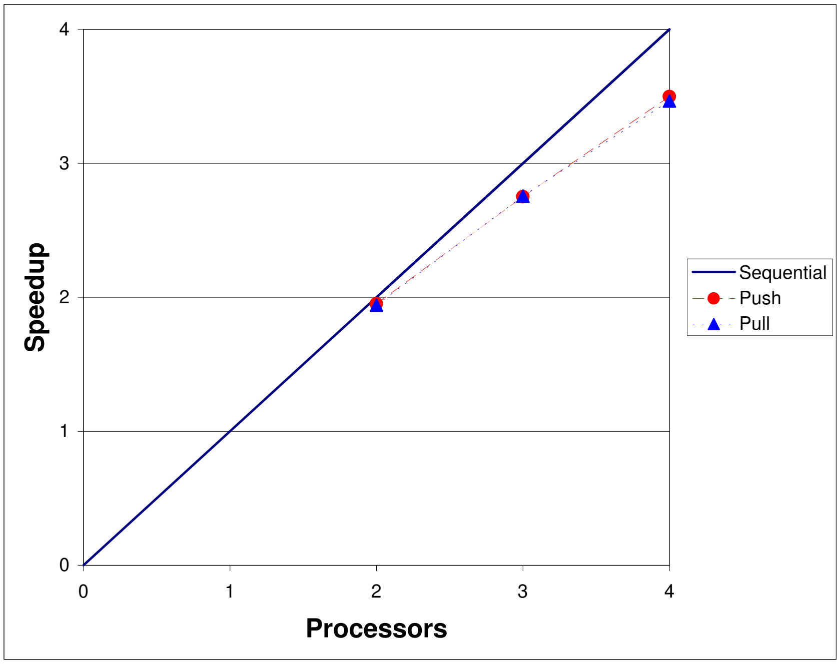


Figure 4.7: Speedups using the Wavefront for sequence alignment.

4.3 Extending CO_2P_3S from Shared-memory to Networks of Workstations

Kai Tan, another graduate student in our research group, has a research goal of extending the CO_2P_3S programming environment from using shared-memory parallel computers to using distributed networks of workstations. One of the tasks that this entails is making modified copies of the existing CO_2P_3S pattern templates (called DMesh, DDistributor, and DPhases) that will run on networks of workstations. MetaCO_2P_3S has been instrumental in this task, enabling Kai Tan to perform the bulk of the conversions with relative ease. This has freed up time for him to concentrate on developing efficient distributed implementations of the pattern templates, and tools to support their use.

Chapter 5

Pattern Template Repositories

The primary goal of this research was to overcome a major limiting factor in the acceptance of template-based parallel programming environments, by providing a tool for extensibility. Sections 1.1 and 2.6 motivated this goal, describing how CO_2P_3S and other template-based parallel programming environments will not become viable until their available templates cover a wide variety of parallel problems. With my introduction of the MetaCO_2P_3S tool, the creation of new pattern templates has become much easier, and the coverage of template-based programming environments can be made arbitrarily wide.

Even though new pattern templates can now be created by the parallel programming community, there must also be a way to share them. To facilitate this sharing, I propose that a central repository be created. Since my pattern templates consist only of XML, Java and image files, they are system-independent, and can easily be packaged in a downloadable format for distribution on the Internet.

In addition to allowing new pattern templates to be shared, another advantage to a central repository is the ability it provides for pattern templates to be refined with new parameters or implementation improvements. The Mesh pattern template went through one such iteration after the discovery of an application that required mesh nodes having eight neighbours instead of four. Our original pattern template only supported mesh nodes with four neighbours. The pattern template was modified by adding a new parameter, and specifying the effect that this new parameter had on the generated framework code. The research being done to extend CO_2P_3S to run on networks of workstations is another example of the usefulness of repositories. Each of the available pattern templates has been copied and modified to support distributed processing.

The creation of a pattern template repository would also extend a challenge to the template-based parallel programming community. Currently, the research being done by different groups on template-based programming environments is almost completely independent. MetaCO_2P_3S provides the the ability for these efforts to be unified. I would like to see this challenge taken up by the community, either through the modification of programming environments to support my pluggable pattern templates, or through the submission of new pattern templates to the shared repository that will eventually make our template-based environments usable and practical.

A number of issues need to be resolved in order to make our vision of a pattern template repository possible, including but not limited to:

- determining what organisation or individual will be responsible for the maintenance and hosting of the repository.
- deciding whether repository access levels are required (e.g. to provide different levels of service to pattern users and pattern designers).
- defining a hierarchy or categorisation for pattern templates.
- providing some means (manual or automatic) for pattern template verification.

5.1 The Generality of My Meta-programming Approach

At one point during the development of the MetaCO_2P_3S tool I made an important discovery about my meta-programming approach. Although the CO_2P_3S environment was built for parallel programming, the generality of the MetaCO_2P_3S tool can make CO_2P_3S independent of the parallel programming domain. My definition of a pattern template is widely applicable, encompassing more than just parallel design patterns. Because of this generality, I surmise that eventually a segmentation of the repository will occur, along with a categorisation of pattern templates.

Chapter 6

Related Research

The history of the CO_2P_3S environment is described in Chapter 2. MacDonald [Mac01] systematically compares CO_2P_3S to a wide variety of parallel programming methodologies. This chapter has a narrower focus, relating the CO_2P_3S metaprogramming extension to related research in high-level extensibility techniques. The parameterisation of design patterns, and their instantiation into object-oriented frameworks, is at the core of CO_2P_3S. Therefore, this chapter also briefly discusses related research in design patterns and frameworks, including solutions to the code generation problem.

One goal of parallel programming systems is simplifying the parallel development process. Pursuing this goal often necessitates compromises elsewhere in a system. One common sacrifice is application performance. However, there is much to judge a system by in addition to its balance of speed and simplicity. The compiled knowledge from almost a decade of research on the predecessors to CO_2P_3S was used to create a list of desirable characteristics for template-based parallel programming systems, as enumerated by Singh [SSS98]. These characteristics were used by MacDonald [Mac01] to evaluate CO_2P_3S, and found that it had done much to advance prior research. One of the major shortcomings that the study identified was the lack of extensibility in CO_2P_3S. The aim of this dissertation was to remove that obstacle, and this chapter compares my approach to the way other systems have dealt with the same problem.

Specific parallel programming languages and parallel libraries are not discussed, since their development process differs dramatically from that of CO_2P_3S. Languages are typically quite general, presenting few restrictions to the programmer. However, they require parallelism to be interwoven with application code, placing the onus on the developer for correct parallelism. Libraries for general parallel communication also suffer from this problem. Domain-specific libraries may be successful at hiding parallel constructs, but they are not applicable to a wide variety of problems. In contrast, template-based systems like CO_2P_3S, although limited by their available templates, are applicable over many domains. Furthermore, development is simplified by separating user application code from parallelism.

6.1 Design Patterns and Frameworks

Design patterns have been defined in Chapter 2.1. Johnson [Joh97] gives a concise definition of a framework. It identifies code and design patterns as the two basic constituents of a framework. In CO_2P_3S, our pattern template abstraction bridges the gap between design patterns and frameworks, and is the mechanism by which we simplify the programming task for our tool's users.

The goals and abstractions of code skeletons are similar to those of our pattern templates. A detailed comparison of the two techniques is presented by Danelutto [Dan01]. However, the tools provided by the skeleton community take a different approach than CO_2P_3S. P^3L [BDO+95] is a parallel programming language based on skeletons. It provides a set of language constructs, or skeletons, that correspond to different types of parallelism. Application-specific code can be supplied to the skeletons, and in turn, the skeletons can be composed by connecting their input and output data streams. The skeletons provided with P^3L are fixed, and cannot be extended.

6.2 Extensible Pattern-based Programming Tools

6.2.1 Generic Programming Environments

A prototype tool for supporting the use of object-oriented patterns is presented by Florijn [FMvW97]. It provides three views into a program: the code, the design, and the design pattern occurrences. After a pattern instance has been added to a program, the tool can generate the necessary classes, but program elements must then be bound to particular roles in the pattern. A refactoring package is provided that allows one to either design a new program using design patterns or document the design patterns in an existing program. The tool does not prevent a user from modifying the pattern semantics in their application. This approach to programming with design patterns is in contrast to CO_2P_3S, which guides programmers through the entire process of pattern template use, and prevents them from modifying pattern semantics. However, one advantage of the tool presented by Florijn [FMvW97] is its tight integration of multiple patterns in a single application. In addition, provisions were made to allow new design patterns to be added to the tool, but the procedure is undocumented.

Two commercial tools that harness design patterns for business application development are available. Both provide extensibility by allowing new design patterns to be added, although no tool is provided to aid in this task. In OmniBuilder [Omn], design patterns represent low-level tasks for user interfaces in business applications. In ModelMaker [Mod], design pattern instances act as macros that insert code into the classes or methods they are told to act upon. Unlike CO_2P_3S, ModelMaker does not separate user code from design pattern implementations.

6.2.2 Parallel Programming Environments

Although many research groups study pattern-based parallel programming environments, few address the need for extensibility. Two such exceptions are DPnDP and Tracs. DPnDP [Siu96, SSGS96] helps to create distributed message-passing programs. Like CO_2P_3S, design patterns in DPnDP are modular, supporting extensibility. However, DPnDP does not provide a tool like MetaCO_2P_3S for creating new patterns, but rather specifies a C++ framework under which patterns can be built. Patterns created using this framework have only a structural specification; all behavioural aspects, such as communication and synchronisation, must be supplied by the DPnDP user. The patterns supplied with DPnDP automatically implement any pattern-specific behaviours. Therefore, new patterns may not have the same level of functionality as those provided with the system, unlike the first-class pattern templates created by MetaCO_2P_3S.

Tracs [BCDP95] allows pattern designers to define architectural models for new patterns using a formal graph to specify task and communication structures. However, the architectural models do not include implementations, so the level of functionality is not comparable to CO_2P_3S.

6.3 Code Generation

Automatic code generation has been studied by many groups with different agendas. In the original CO_2P_3S implementation, the framework code generation took its inspiration from the CORRELATE [MJR+98] and COGENT [BFVY96] macro languages. COGENT was designed as the code generator for a system that automatically generated frameworks for each of the "Gang of Four" design patterns [GHJV95]. Their system did not have any extensibility features.

After the introduction of MetaCO_2P_3S, a more sophisticated code generation mechanism was required to simplify the creation of pattern templates. My idea to use Javadoc for code generation came from Pollack [Pol00]. The current code generator in MetaCO_2P_3S mixes ideas from the CORRELATE approach with Javadoc.

6.4 Pattern Repositories

Pattern repositories are central to the acceptance of pattern-based programming environments like CO_2P_3S. The ACE Catalogue [Sch94] is one such repository, but its patterns centre around network communication mechanisms, and are therefore targeted at an audience with lower level requirements than CO_2P_3S users. The concurrent design patterns in Lea's book [Lea99] also provide lower level parallel constructs.

The Portland Pattern Repository [Cat], although not targeting parallel patterns, has the interesting feature of being built on the WikiWikiWeb system, which allows pattern designers to easily add patterns to the catalogue. None of these repositories store patterns that have the ability to automatically integrate with programming environments, which is one of the key features we are recommending.

6.5 Conclusions

Research groups and commercial interests have started to explore building programming environments using ideas from the design pattern community. Of these, only a handful are targeting the parallel programming community. A major drawback to the majority of these systems is their lack of extensibility. Template-based programming environments are limited in applicability by their available library of templates, unless some provision for extensibility has been made. Only a handful of groups have identified this problem and attempted to solve it. Foremost among these are DPnDP and Tracs. MetaCO_2P_3S is an ambitious effort that has gone beyond previous research. It makes CO_2P_3S the first pattern-based parallel programming system with a tool for extending the environment with new patterns, both indistinguishable in form and equivalent in functionality to the patterns that are predefined.

Chapter 7

Summary and Conclusions

This dissertation has described a research project that enables the creation of parallel design pattern templates. This research stemmed from the need for extensibility in CO_2P_3S. My tool, called MetaCO_2P_3S, allows a pattern designer to create a pattern description and annotated framework template in a standard format that can be imported into CO_2P_3S, or any other compatible parallel programming system.

7.1 Contributions of this Research

There are a number of contributions from this research. Foremost among these is the solution provided by MetaCO_2P_3S to a critical problem in template-based programming systems, namely their lack of available templates. I have defined a system-independent pattern template, including a parameterisable framework template format. My tool creates first-class pattern templates that easily plug in as modules to the CO_2P_3S environment.

Through this extensibility research, I have enabled the creation of a pattern template repository. This extends a challenge to the template-based programming environment community to submit new or improved pattern templates that could be shared with others.

7.2 Ongoing Enhancements to CO_2P_3S and MetaCO_2P_3S

There are a three avenues of new work on the CO_2P_3S environment currently underway in our research group. As introduced in Chapter 4, this work includes an enhancement that will generate frameworks for distributed networks of workstations, in addition to our current shared-memory implementation. We are also using MetaCO_2P_3S to generate new pattern templates for CO_2P_3S. The other body of work that is currently being tackled is the documentation of pattern templates. This enhancement includes the definition of a standard format for documentation, and the integration of the documentation into the CO_2P_3S environment. The documentation will target both CO_2P_3S pattern users, and pattern designers.

7.3 Directions for Future Work

I have identified a number of future enhancements for the MetaCO_2P_3S tool. One is the completion of the user interface to include a wizard that guides pattern designers through the framework template creation process. To tailor a version of MetaCO_2P_3S for parallel pattern template creation, it would also be good to provide a language of parallel primitives that could be used during framework template creation.

Our research group has discussed the implementation of analytical approaches for the parameterisation of framework templates. These could help provide correctness by ensuring that the effects of each possible parameter value are handled correctly.

As introduced in Chapter 5, I propose that a pattern template repository be set up to provide a much-needed resource for the template-based programming community. However, the large number of pattern templates in a central repository introduce another area for future work. Since the status quo requires developers to choose pattern templates for their application with no guidance beyond a design pattern document, a better pattern selection mechanism is needed. Pattern languages [MMS00] may eventually provide a solution to this problem. However, a good short-term fix for the pattern selection problem may lie in segmenting the repository into well-defined categories.

Finally, it would be helpful to gather usability data for the MetaCO_2P_3S tool, to empirically measure the ease with which pattern designers can leverage the tool for their purposes.

Bibliography

[BCDP95] A. Bartoli, P. Corsini, G. Dini, and C. Prete. Graphical Design of Distributed Applications Through Reusable Components. *IEEE Parallel & Distributed Technology*, 3(1):37–51, 1995.

[BDO+95] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P^3L: a Structured High–level Parallel Language, and its Structured Support. *Concurrency: Practice and Experience*, 7(3):225–255, 1995.

[BFVY96] F. Budinsky, M. Finnie, J. Vlissides, and P. Yu. Automatic Code Generation from Design Patterns. *IBM Systems Journal*, 35(2):151–171, 1996.

[Cat] CategoryPattern, Portland Pattern Repository. http://c2.com/ppr/.

[CSS00] K. Charter, J. Schaeffer, and D. Szafron. Sequence Alignment using FastLSA. In *Proc. of the 2000 International Conference on Mathematics and Engineering Techniques in Medicine and Biological Sciences (METMBS'2000)*, pages 239–245, 2000.

[Dan01] M. Danelutto. On Skeletons and Design Patterns. In *Proceedings of PARCO'01 (to appear)*, 2001.

[Doc] Javadoc Doclet API. http://java.sun.com/j2se/1.3/docs/tooldocs/javadoc/doclet/.

[FMvW97] G. Florijn, M. Meijers, and P. van Winsen. Tool Support for Obect-Oriented Patterns. In *Object–Oriented Programming 11th European Conference (ECOOP'97)*, volume 1241, pages 472–495. Springer–Verlag, 1997.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[Jav] Javadoc Tool Home Page. http://java.sun.com/j2se/javadoc/.

[Joh97] R. Johnson. Frameworks = (Components + Patterns). *Communications of the ACM*, 40(10):39–42, October 1997.

[Lea99] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns.* Addison–Wesley, 2nd edition, 1999.

[Mac01] S. MacDonald. *From Patterns to Frameworks to Parallel Programs.* PhD thesis, Department of Computing Science, University of Alberta, November 2001. Available at www.cs.ualberta.ca/∼systems.

[MJR+98] F. Matthijs, W. Joosen, B. Robben, B. Vanhaute, and P. Verbaeten. Multi–level Patterns. In *Object–Oriented Technology (ECOOP'97 Workshop Reader)*, volume 1357 of *Lecture Notes in Computer Science*, pages 112–115. Springer–Verlag, 1998.

[MMS00] B. Massingill, T. Mattson, and B. Sanders. A Pattern Language for Parallel Application Programs. In *European Conference on Parallel Processing*, pages 678–681, 2000.

[Mod] ModelMaker CASE Tool. http://www.modelmaker.demon.nl/mm.htm.

[MSS97] S. MacDonald, J. Schaeffer, and D. Szafron. Pattern-based Object-Oriented Parallel Programming. In *Lecture Notes in Computer Science 1343: 1st International Scientific Computing in Object-Oriented Parallel Environments Conference (ISCOPE '97)*, pages 267–274. Springer-Verlag, December 1997.

[MSS99] S. MacDonald, D. Szafron, and J. Schaeffer. Object-Oriented Pattern-Based Parallel Programming with Automatically Generated Frameworks. In *5th USENIX Conference on Object-Oriented Tools and Systems (COOTS '99)*, pages 29–43, May 1999.

[MSSB00a] S. MacDonald, D. Szafron, J. Schaeffer, and S. Bromling. From Patterns to Frameworks to Parallel Programs. Submitted to Journal of Parallel and Distributed Computing. Available at www.cs.ualberta.ca/~systems, December 2000.

[MSSB00b] S. MacDonald, D. Szafron, J. Schaeffer, and S. Bromling. Generating Parallel Program Frameworks from Parallel Design Patterns. In *Euro-Par 2000, Parallel Processing*, volume 1900 of *Lecture Notes in Computer Science*, pages 95–104. Springer-Verlag, August 2000.

[Omn] OmniBuilder Design Patterns. http://www.omnibuilder.com/overview/design.htm.

[Pol00] M. Pollack. Code Generation using Javadoc. http://www.javaworld.com/javaworld/jw-08-2000/jw-0818-javadoc_p.html, August 2000.

[Sch94] D. Schmidt. The ADAPTIVE Communication Environment: Object-Oriented Network Programming Components for Developing Client/Server Applications. In *Proceedings of the 12th Sun Users Group Conference*, 1994.

[Siu96] S. Siu. Openness and Extensibility in Design–Pattern–Based Programming Systems. Master's thesis, Department of Electrical and Computer Engineering, University of Waterloo, August 1996.

[SSC96] M. Sefika, A. Sane, and R. Campbell. Monitoring Compliance of a Software System with its High–Level Design Models. In *Proceedings of the 18th International Conference on Software Engineering*, pages 387–396. IEEE Computer Society Press, 1996.

[SSG89] A. Singh, J. Schaeffer, and M. Green. Structuring Distributed Algorithms in a Workstation Environment. In *Proceedings of the International Conference on Parallel Processing*, pages 89–97, 1989.

[SSGS96] S. Siu, M. De Simone, D. Goswami, and A. Singh. Design Patterns for Parallel Programming. In *Proceedings of the 1996 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'96)*, pages 230–240, 1996.

[SSLP93] J. Schaeffer, D. Szafron, G. Lobe, and I. Parsons. The Enterprise Model for Developing Distributed Applications. *IEEE Parallel & Distributed Technology*, 1(3):85–96, 1993.

[SSS98] A. Singh, J. Schaeffer, and D. Szafron. Experience with Parallel Programming Using Code Templates. *Concurrency: Practice & Experience*, 10(2):91–120, 1998.

[W3C] World Wide Web Consortium. http://www.w3.org/.

[XML] XML Home Page. http://www.w3.org/XML/.

[XSL] XSL Home Page. http://www.w3.org/Style/XSL/.

Appendix A

Installing CO_2P_3S and $MetaCO_2P_3S$

A.1 Downloading the System

The CO_2P_3S environment is available for download at:

- `http://www.cs.ualberta.ca/~systems/`

After downloading and unpacking the package, the root of the CO_2P_3S installation will be at "`[InstallationDir]/copsProj/`".

A.2 Configuring CO_2P_3S

Copy the "`copsProj/copsrc.xml`" file to your home directory, and rename it to: "`.copsrc.xml`". Edit this file, and set up the following options:

copsInstallationDirectory This should be set to the the directory in which CO_2P_3S is installed. You can use a pathname relative to your home directory. If you installed CO_2P_3S directly in your home account, set this to: "`copsProj`".

userProgramDirectory Set this to the directory in which you want your user programs stored by default. This value can also be set in the preferences dialog of the CO_2P_3S GUI.

defaultEditor Set this to the binary of the editor you wish to use within the CO_2P_3S environment. For instance, "`gvim`" or "`emacs`". This can also be set in the preferences dialog of the CO_2P_3S GUI.

patterns You can leave this setting blank, as it will be modified automatically when new patterns are added.

Java version 1.3 or greater needs to be installed on your system. You need to set up your `CLASSPATH` environment variable to work with CO_2P_3S. Use the following setting, modifying values that refer to your installation directory and the Java installation directory as necessary:

- `CLASSPATH=[instDir]:.:[instDir]/libs/jdom.jar:[instDir]/libs/xerces.jar:[instDir]/libs/xalan.jar:[javaInstDir]/lib/tools.jar:[javaInstDir]/lib/jini-core.jar:[javaInstDir]/lib/jini-ext.jar:$CLASSPATH`

A.3 Building CO_2P_3S

To build CO_2P_3S, change into the "`copsProj/`" installation directory, and execute the "`make`" command.

A.4 Running CO_2P_3S

To run CO_2P_3S, change into the "`copsProj/`" installation directory, and execute the command: "`./runCops`". If any problems occur during the initialisation or execution of the CO_2P_3S environment, inspect the "`copsProj/cops.log`" file. The system log can also be viewed within the CO_2P_3S environment. While running CO_2P_3S, it is advised that the keyboard's NumLock key be turned off, as otherwise it will conflict with certain aspects of GUI operation.

A.5 Adding Supplied Patterns to CO_2P_3S

While running CO_2P_3S patterns can be added to the environment by selecting the menu item: "Environment – Add/Update Pattern". This brings up a file chooser in the "`copsProj/patterns`" directory, allowing the user to import a pattern file, such as "`Mesh.xml`". Once added, patterns will remain in the CO_2P_3S system across executions. Patterns can be removed from the system using the "Environment – Remove Pattern" menu item.

Appendix B

Pattern Template File Formats

B.1 DTD for CO_2P_3S Pattern Template Definitions

```
<!ELEMENT CopsPattern:patternInfo (CopsPattern:patternName,CopsPattern:imagesDir,
    CopsPattern:patternPackage,CopsPattern:constants,CopsPattern:classNames,
    CopsPattern:parameters,CopsPattern:guiInfo)>
<!ATTLIST CopsPattern:patternInfo
      xmlns:CopsPattern CDATA #REQUIRED
>
<!ELEMENT CopsPattern:patternName (#PCDATA)>
<!ELEMENT CopsPattern:imagesDir (#PCDATA)>
<!ELEMENT CopsPattern:patternPackage (#PCDATA)>

<!ELEMENT CopsPattern:constants (CopsPattern:constant)*>
<!ELEMENT CopsPattern:constant (CopsPattern:constantID,CopsPattern:constantValue)>
<!ATTLIST CopsPattern:constant type CDATA "noLongerUsed">
<!ELEMENT CopsPattern:constantID (#PCDATA)>
<!ELEMENT CopsPattern:constantValue (#PCDATA)>

<!ELEMENT CopsPattern:parameters (CopsPattern:parameter*,
    CopsPattern:listParameter*, CopsPattern:extParameter*)>
<!ELEMENT CopsPattern:parameter (CopsPattern:parameterName,
    CopsPattern:parameterMenuText,CopsPattern:parameterDefault?,
    CopsPattern:parameterValidates?,CopsPattern:parameterValues?)>
<!ATTLIST CopsPattern:parameter id CDATA #REQUIRED>
<!ELEMENT CopsPattern:parameterName (#PCDATA)>
<!ELEMENT CopsPattern:parameterMenuText (#PCDATA)>
<!ELEMENT CopsPattern:parameterDefault (#PCDATA)>
<!ELEMENT CopsPattern:parameterValidates EMPTY>
<!ELEMENT CopsPattern:parameterValues (CopsPattern:parameterValue)*>
<!ELEMENT CopsPattern:parameterValue (#PCDATA)>
<!ELEMENT CopsPattern:listParameter (CopsPattern:listParameterName,
    CopsPattern:listParameterMenuText,CopsPattern:listParameterClass,
    CopsPattern:listParameterStrings?,CopsPattern:listParameterEntryClass?)>
<!ATTLIST CopsPattern:listParameter id CDATA #REQUIRED>
<!ELEMENT CopsPattern:listParameterName (#PCDATA)>
<!ELEMENT CopsPattern:listParameterMenuText (#PCDATA)>
<!ELEMENT CopsPattern:listParameterClass (#PCDATA)>
<!ELEMENT CopsPattern:listParameterStrings EMPTY>
<!ELEMENT CopsPattern:listParameterEntryClass (#PCDATA)>
<!ELEMENT CopsPattern:extParameter (CopsPattern:extParameterName,
    CopsPattern:extParameterMenuText,CopsPattern:extParameterClass)>
```

```
<!ATTLIST CopsPattern:extParameter id CDATA #REQUIRED>
<!ELEMENT CopsPattern:extParameterName (#PCDATA)>
<!ELEMENT CopsPattern:extParameterMenuText (#PCDATA)>
<!ELEMENT CopsPattern:extParameterClass (#PCDATA)>

<!ELEMENT CopsPattern:classNames (CopsPattern:userClassName*,
    CopsPattern:frameworkClassName*)>
<!ELEMENT CopsPattern:userClassName (CopsPattern:userClassNameID,
    CopsPattern:userClassNameDefValue?,CopsPattern:userClassNameMenuText,
    CopsPattern:userClassNameDefInsufficient?,CopsPattern:userClassNameIsPatternName?,
    CopsPattern:userClassNameIsTemplate?,CopsPattern:userClassNameExternalRef?)+>
<!ELEMENT CopsPattern:userClassNameID (#PCDATA)>
<!ELEMENT CopsPattern:userClassNameDefValue (#PCDATA)>
<!ELEMENT CopsPattern:userClassNameMenuText (#PCDATA)>
<!ELEMENT CopsPattern:userClassNameDefInsufficient EMPTY>
<!ELEMENT CopsPattern:userClassNameIsPatternName EMPTY>
<!ELEMENT CopsPattern:userClassNameIsTemplate EMPTY>
<!ELEMENT CopsPattern:userClassNameExternalRef EMPTY>
<!ELEMENT CopsPattern:frameworkClassName (CopsPattern:frameworkClassNameID,
    CopsPattern:frameworkClassNameRef,CopsPattern:frameworkClassNameIsTemplate?)+>
<!ELEMENT CopsPattern:frameworkClassNameID (#PCDATA)>
<!ELEMENT CopsPattern:frameworkClassNameRef (#PCDATA)>
<!ELEMENT CopsPattern:frameworkClassNameIsTemplate EMPTY>

<!ELEMENT CopsPattern:guiInfo (CopsPattern:visualElements)>
<!ELEMENT CopsPattern:visualElements (CopsPattern:gElement*, CopsPattern:tElement*)>
<!ELEMENT CopsPattern:gElement (CopsPattern:gElementID,
    CopsPattern:gElementLocationX,CopsPattern:gElementLocationY,
    CopsPattern:gElementImages?,CopsPattern:gElementCurImageParts?)>
<!ELEMENT CopsPattern:tElement (CopsPattern:tElementID,
    CopsPattern:tElementLocationX,CopsPattern:tElementLocationY,
    CopsPattern:tElementMaxLength,CopsPattern:tElementJustification,
    CopsPattern:tElementText,CopsPattern:tElementUpdateType?,
    CopsPattern:tElementUpdateVal?)>
<!ELEMENT CopsPattern:gElementID (#PCDATA)>
<!ELEMENT CopsPattern:gElementLocationX (#PCDATA)>
<!ELEMENT CopsPattern:gElementLocationY (#PCDATA)>
<!ELEMENT CopsPattern:gElementImages (CopsPattern:gElementImage)*>
<!ELEMENT CopsPattern:gElementImage (CopsPattern:gElementImageName,
    CopsPattern:gElementImageLoc)>
<!ELEMENT CopsPattern:gElementImageName (#PCDATA)>
<!ELEMENT CopsPattern:gElementImageLoc (#PCDATA)>
<!ELEMENT CopsPattern:gElementCurImageParts (CopsPattern:gElementCurImagePart*)>
<!ELEMENT CopsPattern:gElementCurImagePart (CopsPattern:gElementCurImagePartVal,
    CopsPattern:gElementCurImagePartType)>
<!ELEMENT CopsPattern:gElementCurImagePartVal (#PCDATA)>
<!ELEMENT CopsPattern:gElementCurImagePartType (#PCDATA)>
<!ELEMENT CopsPattern:tElementID (#PCDATA)>
<!ELEMENT CopsPattern:tElementLocationX (#PCDATA)>
<!ELEMENT CopsPattern:tElementLocationY (#PCDATA)>
<!ELEMENT CopsPattern:tElementMaxLength (#PCDATA)>
<!ELEMENT CopsPattern:tElementText (#PCDATA)>
<!ELEMENT CopsPattern:tElementJustification (#PCDATA)>
<!ELEMENT CopsPattern:tElementUpdateType (#PCDATA)>
<!ELEMENT CopsPattern:tElementUpdateVal (#PCDATA)>
```

B.2 XML Pattern Template Description for **Mesh**

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE CopsPattern:patternInfo SYSTEM "../DTD/CopsPattern.dtd">
<CopsPattern:patternInfo xmlns:CopsPattern=
   "http://www.cs.ualberta.ca/~systems/cops.html">
<CopsPattern:patternName>Mesh</CopsPattern:patternName>
<CopsPattern:imagesDir>IMAGES_DIR + "mesh" + File.separator</CopsPattern:imagesDir>
<CopsPattern:patternPackage>cops.gui.patterns.mesh</CopsPattern:patternPackage>
<CopsPattern:constants>
<CopsPattern:constant>
<CopsPattern:constantID>MESH_NUM_NEIGHBOURS_LBL</CopsPattern:constantID>
<CopsPattern:constantValue>"Set Number of Neighbours"</CopsPattern:constantValue>
</CopsPattern:constant>
<CopsPattern:constant>
<CopsPattern:constantID>MESH_BOUNDARY_CONDS_LBL</CopsPattern:constantID>
<CopsPattern:constantValue>"Set Mesh Boundary Conditions"
   </CopsPattern:constantValue>
</CopsPattern:constant>
<CopsPattern:constant>
<CopsPattern:constantID>MESH_STATE_CLASS_LBL</CopsPattern:constantID>
<CopsPattern:constantValue>"Set Mesh State Class"</CopsPattern:constantValue>
</CopsPattern:constant>
<CopsPattern:constant>
<CopsPattern:constantID>MESH_CLASS_NAME_LBL</CopsPattern:constantID>
<CopsPattern:constantValue>"Set Mesh Class Name"</CopsPattern:constantValue>
</CopsPattern:constant>
<CopsPattern:constant>
<CopsPattern:constantID>MESH_STATE_SUPER_CLASS_LBL</CopsPattern:constantID>
<CopsPattern:constantValue>"Set Mesh State Superclass"</CopsPattern:constantValue>
</CopsPattern:constant>
<CopsPattern:constant>
<CopsPattern:constantID>MESH_ORDERING_LBL</CopsPattern:constantID>
<CopsPattern:constantValue>"Set Mesh Ordering"</CopsPattern:constantValue>
</CopsPattern:constant>
<CopsPattern:constant>
<CopsPattern:constantID>MESH_LBL</CopsPattern:constantID>
<CopsPattern:constantValue>"Mesh"</CopsPattern:constantValue>
</CopsPattern:constant>
<CopsPattern:constant>
<CopsPattern:constantID>MESH_STATE_LBL</CopsPattern:constantID>
<CopsPattern:constantValue>"Mesh State Class"</CopsPattern:constantValue>
</CopsPattern:constant>
<CopsPattern:constant>
<CopsPattern:constantID>MESH_STATE_SUPER_LBL</CopsPattern:constantID>
<CopsPattern:constantValue>"Mesh State Superclass"</CopsPattern:constantValue>
</CopsPattern:constant>
<CopsPattern:constant>
<CopsPattern:constantID>OBJECT_CLASS</CopsPattern:constantID>
<CopsPattern:constantValue>"Object"</CopsPattern:constantValue>
</CopsPattern:constant>
<CopsPattern:constant>
<CopsPattern:constantID>BOUNDARY_LBL</CopsPattern:constantID>
<CopsPattern:constantValue>"Boundary Topology"</CopsPattern:constantValue>
</CopsPattern:constant>
<CopsPattern:constant>
<CopsPattern:constantID>NON_T_BOUNDARY</CopsPattern:constantID>
<CopsPattern:constantValue>"Non"</CopsPattern:constantValue>
```

</CopsPattern:constant>
<CopsPattern:constant>
<CopsPattern:constantID>FULLY_T_BOUNDARY</CopsPattern:constantID>
<CopsPattern:constantValue>"Fully"</CopsPattern:constantValue>
</CopsPattern:constant>
<CopsPattern:constant>
<CopsPattern:constantID>HORIZ_T_BOUNDARY</CopsPattern:constantID>
<CopsPattern:constantValue>"Horizontal"</CopsPattern:constantValue>
</CopsPattern:constant>
<CopsPattern:constant>
<CopsPattern:constantID>VERT_T_BOUNDARY</CopsPattern:constantID>
<CopsPattern:constantValue>"Vertical"</CopsPattern:constantValue>
</CopsPattern:constant>
<CopsPattern:constant>
<CopsPattern:constantID>ORDERING_LBL</CopsPattern:constantID>
<CopsPattern:constantValue>"Computation Order"</CopsPattern:constantValue>
</CopsPattern:constant>
<CopsPattern:constant>
<CopsPattern:constantID>ORDERED</CopsPattern:constantID>
<CopsPattern:constantValue>"Ordered Computation"</CopsPattern:constantValue>
</CopsPattern:constant>
<CopsPattern:constant>
<CopsPattern:constantID>CHAOTIC</CopsPattern:constantID>
<CopsPattern:constantValue>"Chaotic Computation"</CopsPattern:constantValue>
</CopsPattern:constant>
<CopsPattern:constant>
<CopsPattern:constantID>NEIGHBOURS_LBL</CopsPattern:constantID>
<CopsPattern:constantValue>"Number of Neighbours"</CopsPattern:constantValue>
</CopsPattern:constant>
<CopsPattern:constant>
<CopsPattern:constantID>FOUR_POINT</CopsPattern:constantID>
<CopsPattern:constantValue>"4"</CopsPattern:constantValue>
</CopsPattern:constant>
<CopsPattern:constant>
<CopsPattern:constantID>EIGHT_POINT</CopsPattern:constantID>
<CopsPattern:constantValue>"8"</CopsPattern:constantValue>
</CopsPattern:constant>
</CopsPattern:constants>
<CopsPattern:classNames>
<CopsPattern:userClassName>
<CopsPattern:userClassNameID>Collector</CopsPattern:userClassNameID>
<CopsPattern:userClassNameDefValue>MESH_LBL</CopsPattern:userClassNameDefValue>
<CopsPattern:userClassNameMenuText>MESH_CLASS_NAME_LBL
</CopsPattern:userClassNameMenuText>
<CopsPattern:userClassNameDefInsufficient>
</CopsPattern:userClassNameDefInsufficient>
<CopsPattern:userClassNameIsPatternName></CopsPattern:userClassNameIsPatternName>
</CopsPattern:userClassName>
<CopsPattern:userClassName>
<CopsPattern:userClassNameID>Mesh</CopsPattern:userClassNameID>
<CopsPattern:userClassNameDefValue>MESH_STATE_LBL
</CopsPattern:userClassNameDefValue>
<CopsPattern:userClassNameMenuText>MESH_STATE_CLASS_LBL
</CopsPattern:userClassNameMenuText>
<CopsPattern:userClassNameDefInsufficient>
</CopsPattern:userClassNameDefInsufficient>
</CopsPattern:userClassName>
<CopsPattern:userClassName>
<CopsPattern:userClassNameID>MeshStateSuper</CopsPattern:userClassNameID>

```
<CopsPattern:userClassNameDefValue>OBJECT_CLASS
    </CopsPattern:userClassNameDefValue>
<CopsPattern:userClassNameMenuText>MESH_STATE_SUPER_CLASS_LBL
    </CopsPattern:userClassNameMenuText>
<CopsPattern:userClassNameExternalRef></CopsPattern:userClassNameExternalRef>
</CopsPattern:userClassName>
<CopsPattern:frameworkClassName>
<CopsPattern:frameworkClassNameID>Abstract#</CopsPattern:frameworkClassNameID>
<CopsPattern:frameworkClassNameRef>Collector</CopsPattern:frameworkClassNameRef>
</CopsPattern:frameworkClassName>
<CopsPattern:frameworkClassName>
<CopsPattern:frameworkClassNameID>Abstract#</CopsPattern:frameworkClassNameID>
<CopsPattern:frameworkClassNameRef>Mesh</CopsPattern:frameworkClassNameRef>
</CopsPattern:frameworkClassName>
<CopsPattern:frameworkClassName>
<CopsPattern:frameworkClassNameID>Bounded#Array
    </CopsPattern:frameworkClassNameID>
<CopsPattern:frameworkClassNameRef>Mesh</CopsPattern:frameworkClassNameRef>
</CopsPattern:frameworkClassName>
<CopsPattern:frameworkClassName>
<CopsPattern:frameworkClassNameID>#State</CopsPattern:frameworkClassNameID>
<CopsPattern:frameworkClassNameRef>Mesh</CopsPattern:frameworkClassNameRef>
<CopsPattern:frameworkClassNameIsTemplate>
    </CopsPattern:frameworkClassNameIsTemplate>
</CopsPattern:frameworkClassName>
<CopsPattern:frameworkClassName>
<CopsPattern:frameworkClassNameID>#Strategy</CopsPattern:frameworkClassNameID>
<CopsPattern:frameworkClassNameRef>Mesh</CopsPattern:frameworkClassNameRef>
</CopsPattern:frameworkClassName>
</CopsPattern:classNames>
<CopsPattern:parameters>
<CopsPattern:parameter id="ordered_">
<CopsPattern:parameterName>ORDERING_LBL</CopsPattern:parameterName>
<CopsPattern:parameterMenuText>MESH_ORDERING_LBL
    </CopsPattern:parameterMenuText>
<CopsPattern:parameterDefault>ORDERED</CopsPattern:parameterDefault>
<CopsPattern:parameterValidates></CopsPattern:parameterValidates>
<CopsPattern:parameterValues>
<CopsPattern:parameterValue>ORDERED</CopsPattern:parameterValue>
<CopsPattern:parameterValue>CHAOTIC</CopsPattern:parameterValue>
</CopsPattern:parameterValues>
</CopsPattern:parameter>
<CopsPattern:parameter id="boundary_">
<CopsPattern:parameterName>BOUNDARY_LBL</CopsPattern:parameterName>
<CopsPattern:parameterMenuText>MESH_BOUNDARY_CONDS_LBL
    </CopsPattern:parameterMenuText>
<CopsPattern:parameterDefault>NON_T_BOUNDARY</CopsPattern:parameterDefault>
<CopsPattern:parameterValidates></CopsPattern:parameterValidates>
<CopsPattern:parameterValues>
<CopsPattern:parameterValue>NON_T_BOUNDARY</CopsPattern:parameterValue>
<CopsPattern:parameterValue>FULLY_T_BOUNDARY</CopsPattern:parameterValue>
<CopsPattern:parameterValue>HORIZ_T_BOUNDARY</CopsPattern:parameterValue>
<CopsPattern:parameterValue>VERT_T_BOUNDARY</CopsPattern:parameterValue>
</CopsPattern:parameterValues>
</CopsPattern:parameter>
<CopsPattern:parameter id="numNeighbours_">
<CopsPattern:parameterName>NEIGHBOURS_LBL</CopsPattern:parameterName>
<CopsPattern:parameterMenuText>MESH_NUM_NEIGHBOURS_LBL
    </CopsPattern:parameterMenuText>
```

```
<CopsPattern:parameterDefault>FOUR_POINT</CopsPattern:parameterDefault>
<CopsPattern:parameterValidates></CopsPattern:parameterValidates>
<CopsPattern:parameterValues>
<CopsPattern:parameterValue>FOUR_POINT</CopsPattern:parameterValue>
<CopsPattern:parameterValue>EIGHT_POINT</CopsPattern:parameterValue>
</CopsPattern:parameterValues>
</CopsPattern:parameter>
</CopsPattern:parameters>
<CopsPattern:guiInfo>
<CopsPattern:visualElements>
<CopsPattern:gElement>
<CopsPattern:gElementID>meshG</CopsPattern:gElementID>
<CopsPattern:gElementLocationX>10</CopsPattern:gElementLocationX>
<CopsPattern:gElementLocationY>45</CopsPattern:gElementLocationY>
<CopsPattern:gElementImages>
<CopsPattern:gElementImage>
<CopsPattern:gElementImageName>meshGFully4</CopsPattern:gElementImageName>
<CopsPattern:gElementImageLoc>meshGFully4.gif</CopsPattern:gElementImageLoc>
</CopsPattern:gElementImage>
<CopsPattern:gElementImage>
<CopsPattern:gElementImageName>meshGFully8</CopsPattern:gElementImageName>
<CopsPattern:gElementImageLoc>meshGFully8.gif</CopsPattern:gElementImageLoc>
</CopsPattern:gElementImage>
<CopsPattern:gElementImage>
<CopsPattern:gElementImageName>meshGHorizontal4</CopsPattern:gElementImageName>
<CopsPattern:gElementImageLoc>meshGHorizontal4.gif</CopsPattern:gElementImageLoc>
</CopsPattern:gElementImage>
<CopsPattern:gElementImage>
<CopsPattern:gElementImageName>meshGHorizontal8</CopsPattern:gElementImageName>
<CopsPattern:gElementImageLoc>meshGHorizontal8.gif</CopsPattern:gElementImageLoc>
</CopsPattern:gElementImage>
<CopsPattern:gElementImage>
<CopsPattern:gElementImageName>meshGVertical4</CopsPattern:gElementImageName>
<CopsPattern:gElementImageLoc>meshGVertical4.gif</CopsPattern:gElementImageLoc>
</CopsPattern:gElementImage>
<CopsPattern:gElementImage>
<CopsPattern:gElementImageName>meshGVertical8</CopsPattern:gElementImageName>
<CopsPattern:gElementImageLoc>meshGVertical8.gif</CopsPattern:gElementImageLoc>
</CopsPattern:gElementImage>
<CopsPattern:gElementImage>
<CopsPattern:gElementImageName>meshGNon4</CopsPattern:gElementImageName>
<CopsPattern:gElementImageLoc>meshGNon4.gif</CopsPattern:gElementImageLoc>
</CopsPattern:gElementImage>
<CopsPattern:gElementImage>
<CopsPattern:gElementImageName>meshGNon8</CopsPattern:gElementImageName>
<CopsPattern:gElementImageLoc>meshGNon8.gif</CopsPattern:gElementImageLoc>
</CopsPattern:gElementImage>
</CopsPattern:gElementImages>
<CopsPattern:gElementCurImageParts>
<CopsPattern:gElementCurImagePart>
<CopsPattern:gElementCurImagePartVal>"meshG"</CopsPattern:gElementCurImagePartVal>
<CopsPattern:gElementCurImagePartType>String
    </CopsPattern:gElementCurImagePartType>
</CopsPattern:gElementCurImagePart>
<CopsPattern:gElementCurImagePart>
<CopsPattern:gElementCurImagePartVal>boundary_
    </CopsPattern:gElementCurImagePartVal>
<CopsPattern:gElementCurImagePartType>Parameter
    </CopsPattern:gElementCurImagePartType>
```

</CopsPattern:gElementCurImagePart>
<CopsPattern:gElementCurImagePart>
<CopsPattern:gElementCurImagePartVal>numNeighbours_
</CopsPattern:gElementCurImagePartVal>
<CopsPattern:gElementCurImagePartType>Parameter
</CopsPattern:gElementCurImagePartType>
</CopsPattern:gElementCurImagePart>
</CopsPattern:gElementCurImageParts>
</CopsPattern:gElement>
<CopsPattern:tElement>
<CopsPattern:tElementID>meshNameT</CopsPattern:tElementID>
<CopsPattern:tElementLocationX>90</CopsPattern:tElementLocationX>
<CopsPattern:tElementLocationY>40</CopsPattern:tElementLocationY>
<CopsPattern:tElementMaxLength>28</CopsPattern:tElementMaxLength>
<CopsPattern:tElementJustification>CENTER</CopsPattern:tElementJustification>
<CopsPattern:tElementText>MESH_LBL</CopsPattern:tElementText>
<CopsPattern:tElementUpdateType>Class</CopsPattern:tElementUpdateType>
<CopsPattern:tElementUpdateVal>Collector</CopsPattern:tElementUpdateVal>
</CopsPattern:tElement>
<CopsPattern:tElement>
<CopsPattern:tElementID>orderingT</CopsPattern:tElementID>
<CopsPattern:tElementLocationX>90</CopsPattern:tElementLocationX>
<CopsPattern:tElementLocationY>240</CopsPattern:tElementLocationY>
<CopsPattern:tElementMaxLength>28</CopsPattern:tElementMaxLength>
<CopsPattern:tElementJustification>CENTER</CopsPattern:tElementJustification>
<CopsPattern:tElementText>ORDERED</CopsPattern:tElementText>
<CopsPattern:tElementUpdateType>Parameter</CopsPattern:tElementUpdateType>
<CopsPattern:tElementUpdateVal>ordered_</CopsPattern:tElementUpdateVal>
</CopsPattern:tElement>
<CopsPattern:tElement>
<CopsPattern:tElementID>meshStSuperT</CopsPattern:tElementID>
<CopsPattern:tElementLocationX>260</CopsPattern:tElementLocationX>
<CopsPattern:tElementLocationY>140</CopsPattern:tElementLocationY>
<CopsPattern:tElementMaxLength>28</CopsPattern:tElementMaxLength>
<CopsPattern:tElementJustification>CENTER</CopsPattern:tElementJustification>
<CopsPattern:tElementText>MESH_STATE_SUPER_LBL</CopsPattern:tElementText>
<CopsPattern:tElementUpdateType>Class</CopsPattern:tElementUpdateType>
<CopsPattern:tElementUpdateVal>MeshStateSuper</CopsPattern:tElementUpdateVal>
</CopsPattern:tElement>
<CopsPattern:tElement>
<CopsPattern:tElementID>meshStateT</CopsPattern:tElementID>
<CopsPattern:tElementLocationX>260</CopsPattern:tElementLocationX>
<CopsPattern:tElementLocationY>200</CopsPattern:tElementLocationY>
<CopsPattern:tElementMaxLength>28</CopsPattern:tElementMaxLength>
<CopsPattern:tElementJustification>CENTER</CopsPattern:tElementJustification>
<CopsPattern:tElementText>MESH_STATE_LBL</CopsPattern:tElementText>
<CopsPattern:tElementUpdateType>Class</CopsPattern:tElementUpdateType>
<CopsPattern:tElementUpdateVal>Mesh</CopsPattern:tElementUpdateVal>
</CopsPattern:tElement>
</CopsPattern:visualElements>
</CopsPattern:guiInfo>
</CopsPattern:patternInfo>