Deferring Design Pattern Decisions and Automating Structural Pattern Changes using a Design-Pattern-Based Programming System

STEVE MACDONALD
University of Waterloo
KAI TAN, JONATHAN SCHAEFFER, and DUANE SZAFRON
University of Alberta

In the design phase of software development, the designer must make many fundamental design decisions concerning the architecture of the system. Incorrect decisions are relatively easy and inexpensive to fix if caught during the design process, but the difficulty and cost rise significantly if problems are not found until after coding begins. Unfortunately, it is not always possible to find incorrect design decisions during the design phase. To reduce the cost of expensive corrections, it would be useful to have the ability to defer some design decisions as long as possible, even into the coding stage. Failing that, tool support for automating design changes would give more freedom to revisit and change these decisions when needed. This paper shows how a design-pattern-based programming system based on *generative design patterns* can support the deferral of design decisions where possible, and automate changes where necessary. A generative design pattern is a parameterized pattern form that is capable of generating code for different versions of the underlying design pattern. We demonstrate these ideas in the context of a parallel application written with the CO_2P_3S pattern-based parallel programming system. We show that CO_2P_3S can defer the choice of execution architecture (shared-memory or distributed-memory), and can automate several changes to the application structure that would normally be daunting to tackle late in the development cycle. Although we have done this work with a pattern-based parallel programming system, it can be generalized to other domains.

Categories and Subject Descriptors: D.2.6 [**Programming Systems**]: Integrated Environments; D.2.5 [**Coding Tools and Techniques**]: Object-oriented Programming; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*

General Terms: Human Factors, Languages

Additional Key Words and Phrases: Design patterns, object-oriented frameworks, design decisions, software maintenance, parallel programming

---

This work was supported by the Alberta Research Council, the Natural Science and Engineering Research Council of Canada, Alberta's Informatics Circle of Research Excellence, and MACI (Multi-media Advanced Computational Infrastructure).
Authors' addresses: S. MacDonald, David R. Cheriton School of Computer Science, University of Waterloo, 200 University Avenue West, Waterloo, Ontario, CANADA N2L 3G1; K. Tan, J. Schaeffer, and D. Szafron, Department of Computing Science, University of Alberta Edmonton, Alberta, CANADA T6G 2E8. E-mail: stevem@uwaterloo.ca, {cavalier,jonathan,duane}@cs.ualberta.ca. Web: http://plg.uwaterloo.ca/∼stevem, http://www.cs.ualberta.ca/∼{cavalier,jonathan,duane}.
Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 2007

1. INTRODUCTION

During the design phase of software development, the designer must make many fundamental design decisions about the architecture of the system. A design decision was defined in [Jansen and Bosch 2005] as "a description of the set of architectural additions, subtractions, and modifications to the software architecture, the rationale, and the design rules, design constraints, and additional requirements that (partially) realize one or more requirements on a given architecture." These decisions span functional and non-functional aspects, from high-level concerns to low-level details. At the highest level, the designer must decide on the structure of the components in the system to determine how they work to accomplish the desired functionality. At the lowest levels, the designer may need to consider constraints such as performance and the amount of available memory. Part of the challenge of design is that the low-level concerns can impact the high-level decisions.

In response to design decisions, designers may apply architectural styles, design rules and guidelines, and even design patterns. For our purposes, a design pattern is the encapsulation of several design decisions into a larger entity. However, a design pattern must still be adapted for its eventual use. This adaptation process requires additional design decisions to be made. However, the number of decisions that must be made is smaller than would be necessary without the pattern, simplifying the remaining design.

One potential problem in the design process is that some of the above decisions may be made before complete information about the system and its needs is known. This can happen because the application and its problem domain are not completely understood during design, or because the application requirements change over time. The latter can happen during subsequent maintenance of the system. If problems are identified or anticipated during the design phase they can be changed relatively easily, with less cost in development and retesting. However, making changes later in the development cycle can be more problematic and expensive. This fact is borne out by experience and is part of software engineering texts [Schach 2007]. Further, such changes accumulate and maintenance becomes increasingly difficult [van Gurp and Bosch 2002]. Though we would like to find and address these problems during design, it is not always possible to avoid the need for changes after development has started.

There are two options to help address the problems associated with design decision changes. The first option is to defer the decisions for as long as possible, increasing the chance that the sufficient information will be available when the decision is finally made. However, not all design decisions can be deferred; some basic decisions must be made before initial coding can start. Thus, the second option to address design decision changes is to provide tool support that automates such changes, making it easy to revisit and modify the application design for later change.

This paper demonstrates how a design-pattern-based programming tool based on framework generation can support both of these options, deferring some design decisions until late in the development cycle and automating changes for decisions that cannot be deferred. More specifically, we look at how a pattern-based programming tool can allow a user to defer or change the design decisions concerning the process of adapting the pattern to its use in an application. As noted above, the adaptation of design patterns is only one source of design decisions. We focus on these decisions because design patterns are the focus of our tool support and not because pattern adaptation is inherently more valuable than any other source of design decision.

To show that this approach is feasible, we use an example from the parallel programming domain. The Parallel Design Patterns (PDP) process is a pattern-based approach for creating parallel applications based on *generative design patterns*. Generative design patterns are an implementation-time construct based on an underlying design pattern. A generative pattern generates object-oriented framework code from a parameterized design pattern description, in this case indicating the parallel structure of the application. CO_2P_3S (Correct Object-Oriented Pattern-based Parallel Programming System, pronounced "cops") implements this process for parallel Java applications that can be run on shared-memory and distributed-memory architectures. We use a reaction-diffusion simulation example to show how CO_2P_3S can support deferring some design decisions and automating changes for others. We can quickly change several key characteristics of our parallel application in a matter of minutes. The changes impact the communication structure of the application and even port the application from a shared-memory multi-threaded architecture to a distributed-memory networked architecture with almost no impact on the application code. Although the programming system and example come from the parallel domain, the ideas and techniques presented here can be generalized to other problem domains.

Earlier papers on CO_2P_3S concentrated on initial program development. They showed how to build a parallel application, using the generative design patterns supported by the tool to generate a framework and writing an application using that framework. This paper differs in that it considers the maintenance of CO_2P_3S applications. More specifically, this paper considers the role that CO_2P_3S and its supported generative design patterns can play when changing an application after initial development. Such changes may be in response to changes in the structure of the computation and to changes in the execution environment.

As noted above, design patterns are not the only source of design decisions in an application. We focus on patterns here because our programming tool generates object-oriented frameworks from a generative pattern. These frameworks allow us to simplify the problem by isolating the pattern-specific structural code from the application-specific user code. The former code is affected by the pattern adaptation process, but this adaptation can also affect the user code and, more problematically, the interface between the structural and application code.

As a result, we can categorize design decisions into *interface-neutral decisions* and *interface-affecting decisions*. Interface-neutral decisions affect only the implementation of the pattern structure behind a stable interface. These decisions are amenable to deferral; the rest of the system can target the stable interface and the implementation can be determined later, after sufficient information is available, and can be changed if necessary. In contrast, interface-affecting decisions affect both the pattern structure and the framework interface to the application code. The application code must be changed to use the new interface. Refactoring tools may suffice for simple cases, but more complex cases will require better support. In particular, when adding new method parameters, it is difficult for a refactoring tool to change existing call sites to use call-site-specific variables for the new parameters. Instead, a generic default value provided by the user must be used. Automation for interface-affecting changes is especially important because these decisions cannot be deferred. Coding cannot begin until initial values for these decisions are made, as components must have an initial interface. Once made, these decisions can be difficult to change [Klusener et al. 2005]. Tool support that automates interface and structural code changes makes it easy to revisit and modify the design of an application, making applications more amenable to later changes.

This paper is organized as follows. Section 2 discusses the impact of design decisions and design patterns on applications, as well as limitations in pattern implementations. Section 3 describes our expectations for tool support for design-pattern-based programming. Section 4 details the PDP process and introduces the CO_2P_3S parallel programming system. We present an example reaction-diffusion application to show that CO_2P_3S can defer some design decisions and automate changes in others in Section 5. Section 6 provides some brief performance results from CO_2P_3S programs. Related research is described in Section 7 and the paper concludes with Section 8.

2. DESIGN DECISIONS AND THEIR EFFECTS

2.1 The Impact of Early Design Decisions

During the design process, the designer must make many decisions that will determine the capabilities and structure of the software. These decisions will be influenced by a number of factors, both functional and non-functional, high-level and low-level. The intent of the design phase is to assess all of these factors and create a system that addresses them.

High-level decisions revolve around the basic architecture of the system, which is the set of objects in the system, the relationships and distribution of responsibilities among them, and their interfaces. These kinds of decisions must be made before the implementation phase can begin.

Designing software is difficult for many reasons. One difficulty in design is that high-level decisions are influenced by low-level, non-functional concerns. A good example is performance considerations in a distributed object system. It is common for a sequence of fine-grained method calls to be replaced with one larger method to reduce networking overhead (see Figure 1), altering the interface to an object in response to performance concerns. Another difficulty is that design decisions may be made before complete application needs are properly determined. The designer must make assumptions about current and future application needs to arrive at an initial design. However, these assumptions may change over time, and thus need to be revisited and changed. This is part of the reason why the design process is iterative.

Unfortunately, not all design problems are uncovered in the design process. In some cases, it is not until the implementation has started that design assumptions prove to be incorrect. This can happen for a number of reasons. The decision may have been made too early, before the problem was understood well enough to make the correct choice. The requirements may have changed in response to new user requirements. The application may now be in its maintenance phase and requires changes that were not foreseen in the original design and implementation. It is also possible that some of the non-functional assumptions have changed. The execution architecture may have changed, impacting performance requirements and requiring hardware architecture assumptions to be revisited. The key is that changing the design once coding has begun is much more expensive. The (now incorrect) assumptions and design choices are embedded into the artifacts that make up the software. Locating these assumptions in source code can be difficult, and changing them even harder. If high-level decisions are revisited, the scope of the changes throughout the system can be large, involving many classes that must be modified.

In some cases, the scope of changes can be minimized through the use of encapsulation, modularity, and information hiding. If the interface between a module and its outside environment does not change, its implementation can be changed more easily. However, if

```
public class Client
{
  public void client() {
    // Three (potentially
    // expensive) remote calls.
    server.remoteMethod1();
    server.remoteMethod2();
    server.remoteMethod3();
  }
}

public class Server
{
  public void remoteMethod1() {
    . . .
  }
  public void remoteMethod2() {
    . . .
  }
  public void remoteMethod3() {
    . . .
  }
}
```

(a) A client calling three remote methods using three remote calls.

```
public class Client
{
  public void client() {
    // Amalgamate the three
    // remote calls into one.
    server.remoteMethod();
  }
}

public class Server
{
  // Call the three methods
  // with one remote call.
  public void remoteMethod() {
    this.remoteMethod1();
    this.remoteMethod2();
    this.remoteMethod3();
  }

  public void remoteMethod1() {
    . . .
  }
  public void remoteMethod2() {
    . . .
  }
  public void remoteMethod3() {
    . . .
  }
}
```

(b) A client calling three methods with one remote call.

Fig. 1: Optimizing remote method invocation.

that module consists of many internal classes, design changes can still be a daunting and expensive task.

2.2 The Role of Design Patterns

A design pattern is normally described as a solution to a recurring design problem in a particular context [Gamma et al. 1994]. To this definition, we add a crucial point: a design pattern is a fluid, adaptable solution. It is key to recognize that a pattern does not represent a single, static solution to problems within its context. A design pattern is an outline of a solution that must be adapted for its eventual use. A pattern is better thought of as a family of design solutions. A pattern description describes the basic structure of the family members and outlines common ways this structure can be specialized for different applications.

The first, and easiest, argument that a single representation is insufficient for most patterns is that the class names and method signatures are, with few exceptions, specific to the application and not generic to the pattern. It is not feasible to divorce a pattern from the application context in which it is applied.

A second, and better, argument can be made by noting that most design pattern descriptions include a discussion on implementation alternatives that adapt the basic pattern solution for a variety of application needs or even different programming languages. The "Implementation" section for the patterns in [Gamma et al. 1994] is one example. For example, in the Composite pattern [Gamma et al. 1994], the methods for managing the child objects of a node can be defined in one of two places: in the class representing the composite objects or in the abstract superclass that defines the interface for both composite and leaf objects. The former alternative chooses safety, only allowing child management on objects that can have child objects. The latter alternative chooses transparency, allowing child management methods to be invoked on any object in the composite. However, it also requires leaf objects, which do not have children, to provide an implementation of management methods. Another good example of different versions of a pattern is the Adapter pattern [Gamma et al. 1994]. The first version is an object-based version based on delegation, and can be implemented in any programming language. The second version is a class-based version based on multiple inheritance of implementation, limiting it to languages like C++ and Eiffel. Though both Adapter solutions are for the same problem context, they are different in structure and approach.

In terms of design, we can view a design pattern as the encapsulation of several design decisions into a larger entity that is easier to apply. This view is shared by [Jansen and Bosch 2005]. A pattern describes the basic structural elements and their interactions that solve a design problem within its context. However, this is an outline of the solution; the structure must be adapted to the specific characteristics of the application. Or rather, the most appropriate member of the pattern family must be selected based on the characteristics of the application. Adapting the pattern requires some additional design decisions to be made. However, the number of decisions that remain is smaller because the pattern encompasses many of the important decisions, simplifying the remaining design. It is important to note that the decisions made during the adaptation process may change the high-level functional characteristics of the design (the set of objects and their interface). The Composite and Adapter patterns described above show examples of this.

This adaptation process may also alter the low-level non-functional characteristics of the design. Most of the design patterns for Java 2 Enterprise Edition web-based applications have the goal of improving performance in their distributed execution environment [Alur et al. 2003]. The patterns in [Schmidt et al. 2000] provide efficient architectures for both networked and concurrent programs. These non-functional requirements are also an important part of application development that must be considered during the design process.

Because design patterns are general solutions, they are presented as documents on paper or as Web pages. The patterns are applied at design time. However, as noted, the use of a pattern requires that it be adapted for the application in which it is being applied. The design decisions made during the adaptation process can also be made too early and can be costly to change after coding begins.

2.3 Limitations in Design Pattern Implementation

The basic problem with design patterns is that they are abstract, reusable designs that exist only as documents. There is no concrete representation that can be reused - a pattern must be implemented each time it is used in a design. At least part of the reason for this is that most patterns take on application-specific characteristics. This can be as simple as

application interfaces for the objects making up the pattern, or as complex as structural pattern variations that redistribute responsibilities across objects. The Adapter and Composite examples in the previous section illustrate this problem.

An obvious solution is to provide an object-oriented framework that implements the design pattern. In this sense, a framework is a basic outline of a particular application, such as a graphical user interface [Johnson and Foote 1988]. The framework implements the flow of control through the classes and objects that make up the application, and application logic is inserted in *hook methods*. The framework could capture the commonalities in the different pattern variations. However, the need for a pattern to be fluid and adaptable prevents a generic framework implementation of many patterns [MacDonald et al. 2002]. Though there is some similarity between pattern implementations in different applications, it cannot be exploited with a single, static framework.

Consider, for example, the implementation of the composite classes in the Composite pattern. The primary responsibility for the methods in this class is to iterate over its child objects and invoke the same method on each of them. Implementing this is straightforward, but the methods and their signatures are not generic but rather application-specific. This prevents a single, efficient framework implementation. While it would be possible to use facilities like Java Reflection to provide a single implementation, this solution would perform poorly.

The structural variations of a pattern can also be difficult to abstract into a single framework. For example, the choices for child management methods in the Composite example show pattern variations that redistribute responsibilities across classes. The Adapter example showed variations that can take two fundamentally different approaches to the implementation. Though these are perhaps extreme examples, they show that frameworks do not adequately capture the flexibility and adaptability that is essential to capture the full utility of design patterns.

3. TOOL SUPPORT FOR DESIGN-PATTERN-BASED PROGRAMMING

Given the role of design patterns as outlined in the previous section, we can now consider the kind of support we expect from a design-pattern-based programming system. We concentrate on the development stage of an application that is using design patterns and not on design tools. Design tools and methodologies are clearly important and their inclusion in a pattern-based development tool would be beneficial, but they are not the focus of this work. One example of such work is the parallel design pattern language in [Mattson et al. 2004], which helps a designer analyze the characteristics of a problem and leads to one or more parallel design patterns that can be applied.

It also must be noted that the tool support for design-pattern-based programming that we outline in this section is independent of application domain and implementation language. However, the set of design patterns supported by a particular tool can (and often does) focus on a specific problem domain.

At the most basic level, tool support for design-pattern-based programming should elevate patterns from a purely design artifact to a more concrete form that includes source code. To this, we add the requirement that a tool cannot support only static versions of design patterns. Adaptability is a fundamental property of design patterns that a programming tool must support. Static patterns will not be applicable to a large number of applications, reducing the usefulness of any pattern-based programming tool based on them.

Table I: A summary of the 13 desirable characteristics of a pattern-based parallel programming tool.

Category	Characteristics
Structuring the Parallelism	Separation of pattern specification from application code Hierarchical resolution of parallelism, using patterns within patterns Mutually-independent generative patterns Extensible set of generative patterns Large collection of useful patterns Openness, to use low-level code and mechanisms in application code
Programming Concerns	Correctness (*i.e.*, programs guaranteed to be free of deadlock) Use an existing language, with no changes to syntax or semantics Non-intrusiveness - programming model should not alter coding style
User Satisfaction	Performance (within limits of selected patterns and architecture) Support tools for design, coding, debugging, and performance monitoring Tool usability, supported by studies where possible Portability to move applications to different architectures

Clearly this is not a complete set of characteristics that would be desirable in a pattern-based programming system. A more thorough set of 13 characteristics for programming tools in the parallel domain have been proposed [Singh et al. 1998], including extensibility (allowing users to add new patterns), openness (providing complete application and runtime source code to the user), and others. The complete set of characteristics are summarized in Table I. These characteristics are equally applicable to any application domain. However, without the basic support outlined above these additional characteristics are meaningless, so we will focus on basic design pattern support.

The most basic support for a design-pattern-based tool is to provide a new pattern form that includes source code to help with the implementation phase. Even ignoring structural pattern variations, a single static piece of code would not be a complete solution as it would not allow the pattern implementation to use application-specific interfaces. To see why this is important, we can break the source code for a pattern implementation into three parts:

(1) *Physically-common* code. This code is identical across different pattern applications. The code for managing dependent objects in an Observer pattern [Gamma et al. 1994] falls into this category.

(2) *Logically-common* code. The purpose of this code is common across all pattern implementations, but the code is not static. However, the implementation can be derived from the application context, where this context is typically an application-specific interface. The methods in the composite objects in the Composite pattern are examples of logically-common code. These methods iterate over the children and invoke the same method on each. However, these methods and their implementation require knowledge about the application-specific interface supported by the composite.

(3) *Application-specific* code. This is application code that appears in the classes that make up the pattern. The implementation of the leaf classes in the Composite pattern is a good example.

A static version of pattern code would be capable of capturing the physically-common code in a pattern implementation. Unfortunately, logically-common code seems to be the most common in most design patterns. Since the purpose and structure of logically-common code is well-defined in a pattern, this code can be generated automatically with a sufficiently powerful code generation tool that is given appropriate application context information.

It is essential that any tool support for design patterns capture their adaptable nature. While such a system could try to address the problem by making each pattern variation its own entity, the resulting explosion in the design pattern space would almost certainly be too much for users. The utility of such a programming tool would be too small to be of practical use.

To capture the adaptable quality of a pattern, we suggest the use of an intermediate pattern form. This new form would bridge the abstract design pattern and the concrete code that implements it, and support a subset of the family of design pattern solutions. It would represent the basic design pattern, and accommodate adaptability by exporting a set of options whose values are set by the user. These options customize the basic pattern, adapting it for the application. Or, rather, the options correspond to a subset of the remaining design decisions that must be made after the pattern is applied. Once the intermediate form matches the application needs, it can be given to a code generator to produce an implementation of the selected pattern variation.

The pattern implementation code can be generated as an object-oriented framework. While we found that frameworks are not a general solution to the problem of implementing design patterns, they are still well-suited for the code generated for an intermediate pattern form. A framework clearly separates application-independent structural code from application-specific logic. This separation is a desirable property of the generated code. This allows the user to write application code without needing details about the structural code, which is advantageous when the structure is complex. In addition, this separation means that the structure can be changed with little impact on the application code, a use of the Generation Gap pattern [Vlissides 1998].

Given that we have assembled context information through the option values in the intermediate pattern form, the generated frameworks can be specific to the selected pattern variation. Since the code generation is guided by the option values, it is possible to include both the physically-common and logically-common code. The generated framework code is smaller and easier to work with, and may also have better performance than more general code. In addition, we are not restricted to generating only the framework code. A Façade pattern [Gamma et al. 1994] can also be included in the generated code. This Façade can create and assemble the framework objects that make up the pattern, rather than requiring the user to do so. It can also provide a simple, high-level interface to the framework.

To summarize, we suggest the use of an intermediate pattern form for use in a design-pattern-based programming system. This form provides options so that it can be adapted, where these options correspond to the design decisions that must still be made after the application of the pattern. Once the options are set, the form can be used to generate object-oriented framework code. The structural code of this framework is generated based on the option values, and consists of the physically-common and logically-common code.

The intermediate form also has the benefit of distinguishing between the general, abstract design pattern and the entity supported in the programming tool. Otherwise, the term “design pattern” becomes overloaded, which invariably leads to confusion.

The need for an adaptable design pattern form leads to another concern. We must balance usability against generality. We can provide more options in the intermediate form to support a larger subset of the family of solutions for the underlying design pattern. However, a larger number of options can make the intermediate form more difficult to use. On the other hand, reducing the number of options reduces the potential pattern structures that can be specified as some aspects of the pattern cannot be customized. This can limit the

usefulness of the intermediate form or can result in less efficient framework code. Striking this balance must be done on a pattern-by-pattern basis.

4. THE PARALLEL DESIGN PATTERNS PROCESS

This section describes the Parallel Design Patterns (PDP) process, a pattern-based approach to building parallel applications [MacDonald 2002; MacDonald et al. 2002]. The process is based on a layered approach that provides three distinct abstractions to programmers: the Patterns Layer, the Intermediate Code Layer (not to be confused with intermediate code used in compilers), and the Native Code Layer. The Patterns Layer provides support for correct parallel programming based on a flexible design pattern description of the parallel structure. The structure is specified by selecting and specializing a set of *generative design patterns*, which corresponds to our intermediate pattern form. A generative design pattern is an implementation-time construct that is based on an underlying design pattern. To capture the flexibility of the underlying pattern, a generative pattern exports a set of options that customize the pattern for its application. In the PDP process, once the options for a generative pattern match the application needs, the generative pattern is used to generate a framework that correctly implements the selected structure, including all of the necessary parallel code. This structure is hidden from the user at the Patterns Layer. The user completes the application by providing sequential hook method bodies. The Intermediate Code and Native Code layers gradually expose the structural details of the generated framework at different levels of abstraction. They allow the code to be tuned to remove performance bottlenecks or modified to implement a variation of the selected pattern.

The PDP process consists of five steps, the first three of which are required and the last two optional:

(1) Identify the parallel design patterns that are required to parallelize the application and select the corresponding generative design patterns.

(2) Supply the application-specific options for the selected generative patterns and generate the framework code.

(3) Implement the application-specific sequential hook methods in the generated frameworks, as well as any other application code that is needed. The Patterns Layer is now complete. Check that the result is a semantically correct parallel program. If not, return to Step 1.

(4) If the performance of the parallel application is not satisfactory, use the facilities of the Intermediate Code Layer to examine the structural framework code. This code contains high-level synchronization and communication primitives. Any primitives that are not needed for the particular application can be removed. The remaining primitives can be optimized (for instance, by changing their location in the framework).

(5) If the performance of the parallel application is still not satisfactory, specialize the implementation of the primitives at the Native Code Layer. The new implementation can take into account the characteristics of both the application and the target execution environment (machine architecture, available system libraries, *etc*.).

Though the PDP process was initially conceived in the context of parallel programming, it has since become clear that it applies to many programming domains where design patterns are useful. The ideas have been applied to sequential design patterns from [Gamma et al. 1994] in [MacDonald et al. 2002], the development of network server applications

in [Guo 2003], and even scripting for commercial computer games in [McNaughton et al. 2003].

4.1 CO_2P_3S: The PDP Process Realized

CO_2P_3S (Correct Object-Oriented Pattern-based Parallel Programming System, pronounced "cops") is a concrete realization of the PDP process. CO_2P_3S generates Java frameworks for several common parallel structures, both shared-memory code using threads and distributed-memory code [Tan 2003], the latter using a combination of Jini [Sun Microsystems Inc. 2001] and Java Remote Method Invocation (RMI). Note that CO_2P_3S is our illustration of the PDP process and is not the only possible implementation. The process is independent of programming language and execution architecture. It is intended as a guideline for future design-pattern-based tools, which may support a different set of generative patterns or even have generative patterns with different option values for customization and different hook methods for application code.

A brief overview of the Patterns Layer of the PDP process in CO_2P_3S is shown in Figure 2.

The CO_2P_3S system is available for download at

`http://www.cs.ualberta.ca/~systems/cops/index.html`

5. REACTION-DIFFUSION TEXTURE GENERATION IN CO_2P_3S

In this section, we will briefly describe an example application that was implemented using the CO_2P_3S system. The application is a computer graphics texture generation program based on a reaction-diffusion simulation [Witkin and Kass 1991].

We will start by describing the example application. Next, we will briefly describe the initial program development process in CO_2P_3S. Further details on this process can be found in [MacDonald 2002; MacDonald et al. 2002; MacDonald et al. 2000]. Once we have an initial implementation of the application, we will then focus on the contribution of this paper: how a design-pattern-based tool like CO_2P_3S can support changes to design decisions using a combination of deferral and automation. We start by describing the kinds of design decision changes that a user could make to the reaction-diffusion application, in particular examining changes to the computation structure and execution architecture. We use these changes to highlight how CO_2P_3S supports the necessary changes.

5.1 Reaction-Diffusion Texture Generation

The reaction-diffusion application simulates two interacting chemicals called *morphogens* as they simultaneously diffuse across a two-dimensional surface and interact with one another over time. The simulation is actually solving a non-linear set of partial differential equations using finite differences. The surface is discretized into an array of cells, each with a concentration of each morphogen. Time is advanced in small steps, and at each step the concentration of each morphogen in each cell is calculated using the concentrations in neighbouring cells. This computation is repeated until the concentrations converge to a final value (when the difference between successive iterations falls below a threshold for each cell) or a maximum number of iterations is reached.

With the correct computation function, simulation parameters, and initial conditions, this simulation can generate textures that resemble zebra stripes, leopard spots, and other natural phenomena. The textures that we will generate resemble zebra stripes. An example texture is shown in Figure 3.

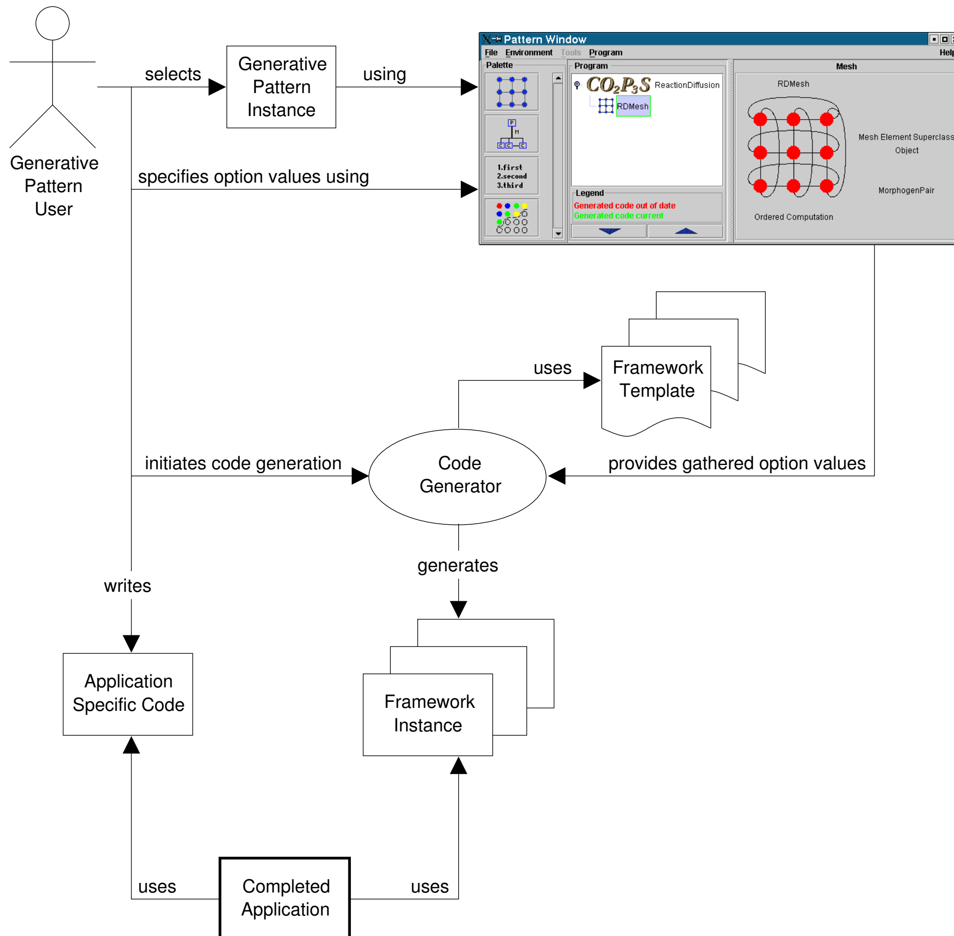


Fig. 2: An overview of the PDP process in the CO_2P_3S development system.

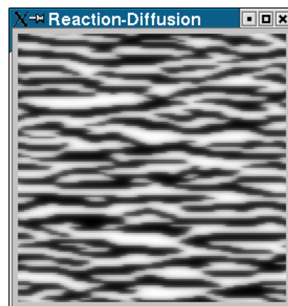


Fig. 3: An example texture generated by the reaction-diffusion example program.

5.2 Reaction-Diffusion in CO_2P_3S

The first step in the PDP process is to identify the design patterns that best suit the problem. The reaction-diffusion application structure is well-known in the parallel programming community as a mesh computation, operating on regular two-dimensional data in this case. To parallelize a mesh computation, the data is split into partitions, which are distributed across a set of processors.[1] Each processor computes new values for its local partition of the data. However, there are data dependencies between partitions, meaning that the processors cannot work completely independently. Computing a new value for an element requires data from the neighbouring elements, so elements on the boundary of a partition will require data from the boundary of adjacent partitions on different processors. This boundary exchange for neighbour data requires interprocessor communication and defines the communication structure of the mesh computation.

However, the mesh is more than just this computation and communication structure. There are other structural aspects of the mesh that can vary, of which we will consider two. The first aspect is the communication pattern of the mesh, which is the set of processors with which a given processor must exchange partition boundaries. This pattern is a function of two properties of the mesh computation: the *stencil* of the mesh and the *topology* of the data. The stencil is the set of neighbouring elements used to compute the new value for a mesh element, which will determine adjacency between the partitions. The topology is the shape of the data. In some applications, the data may wrap around rather than having hard edges, again determining adjacency between partitions.

The second structural aspect to consider in mesh computations is the synchronization structure of the mesh. This aspect determines the timing relationship between processors executing the computation. Some applications require that each iteration use neighbouring data from the previous iteration, so a processor cannot start the next iteration on its partition until all processors have completed the previous iteration and the boundaries have been exchanged. This synchronization level, which we will call a *synchronous iteration*, produces deterministic results at the expense of added synchronization. Other applications can use *asynchronous iteration*, allowing processors to execute at their own pace, using old neighbouring data if necessary. These applications may finish more quickly at the expense of making it more difficult to determine when the data has properly converged.

Having identified the most appropriate design pattern for the problem, we complete the first step in the PDP process by selecting the corresponding generative design pattern. Figure 4 shows this selection in the CO_2P_3S programming system. The supported generative design patterns are shown in the palette on the left. CO_2P_3S supports a Two-Dimensional Mesh generative pattern (simply the Mesh generative pattern from this point onward), the topmost pattern in the palette. It has already been selected and appears on the right-hand side. The middle pane shows the set of generative patterns that make up the current program.

The second step in the PDP process is to supply application-specific option values for the generative patterns. The right-hand pane in the CO_2P_3S interface provides dialogs for entering these values, available through popup menus, shown in the right-hand pane of Figure 4. The resulting pattern instance is shown so the user can quickly see the option values. The Mesh generative pattern options and their values for the reaction-diffusion

[1] We use the generic term *processor* to denote a process or thread assigned to a physical CPU.

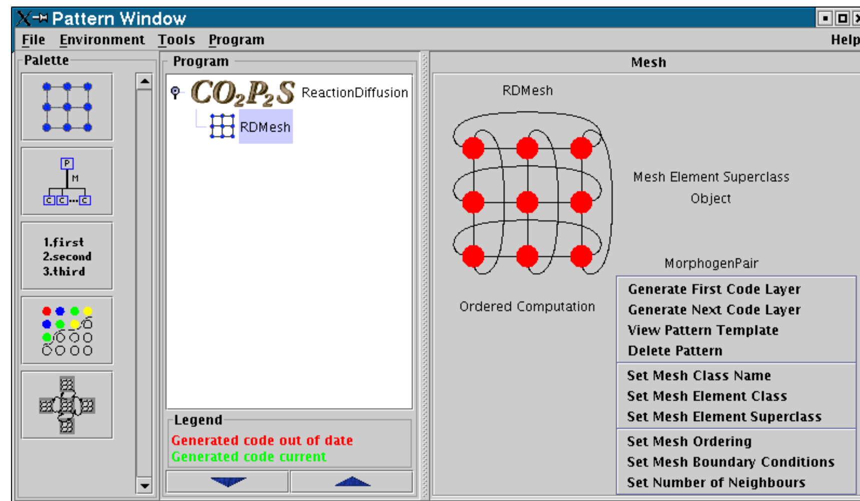


Fig. 4: The CO_2P_3S parallel programming system user interface.

application are:

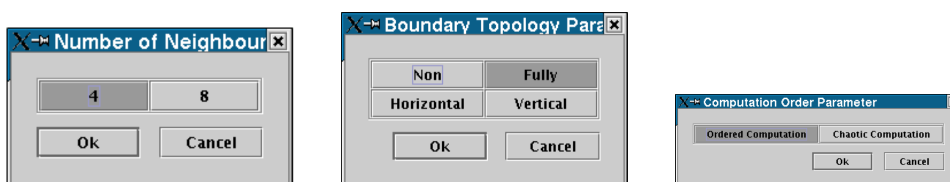
—The name of the class representing the mesh. This class acts as the Façade for the Mesh framework, as mentioned in Section 3. As shown in the middle pane and top of the right-hand pane, we use `RDMesh` for this example.

—The name of the mesh element class and its type (object or primitive). The framework creates a two-dimensional data structure and populates it with values of this type. We use the class name `MorphogenPair` with type object. Each instance of this class holds a pair of morphogen objects and other simulation constants for each cell in the mesh. An application-specific superclass of this mesh element class can also be provided; there is no such superclass in this example. It is also possible to specify that the internal mesh data should be populated with one of the primitive types in Java (`int`, `double`, *etc.*) and create the mesh element class as a set of static methods that operate on primitive data, since this can improve performance where objects are not necessary.

—The number of neighbours. This determines the stencil that is used to calculate new values for an element in every mesh iteration. In this specific instance of the reaction-diffusion simulation, the morphogens diffuse horizontally and vertically, so we use a four-point stencil. An eight-point stencil, including elements on the diagonals, is also available for more complex simulations. The dialog for specifying this parameter is shown in Figure 5(a).

—The topology of the data. This application uses a fully-toroidal mesh, where edges wrap both horizontally and vertically. The data forms a torus (or donut) shape with no edges. This topology allows the resulting texture to be tiled across a large display without any discernible edges. Horizontal-toroidal, vertical-toroidal, and non-toroidal topologies are also supported. The dialog for entering this information is shown in Figure 5(b).

—The level of synchronization. This problem uses synchronous iteration, which meets the original problem specification. The dialog for entering this parameter value is shown in Figure 5(c).



(a) CO_2P_3S dialog for specifying stencil.

(b) CO_2P_3S dialog for specifying boundary conditions.

(c) CO_2P_3S dialog for specifying synchronization level.

Fig. 5: CO_2P_3S dialogs for entering 2D Mesh parameter values.

Once these options are provided, the generative design pattern is used to generate a complete framework. In CO_2P_3S, this is done with the "Generate First Code Layer" item in the popup layer (shown in Figure 4). Our code generation process is guided by the generative pattern option values, so the framework implements the requested member of the pattern family. For our example, the generated Mesh framework is specific to the option values given above. The code generation process is briefly outlined in Section 5.6.

At the Patterns Layer, the structural part of the framework is hidden from the user. Instead, the user can only access the generated classes that export hook methods. In CO_2P_3S, this is enforced through tool support. In the Mesh framework, the only class that exports hook methods is the mesh element class. This is the `MorphogenPair` class in our example. At the Patterns Layer, this class can only be accessed using a modified HTML viewer shown in Figure 6. This tool support removes several sources of user error. Only the hook method bodies can be modified, by selecting a link to bring up the code editor shown in Figure 7. The user cannot modify the method signatures since the structural framework code relies on these signatures as the interface to the application-specific code. This tool support removes several sources of user error. The user also cannot accidentally introduce errors into the concurrent framework code since it is not accessible.

To complete the application at the Patterns Layer, the user inserts application code into the hook methods exported by the framework. This code can take advantage of code from the sequential version of the program since hook method code does not include any concurrency or synchronization code. The details on the hook methods and how they are used to construct a complete application is outlined in other papers [MacDonald et al. 2002; MacDonald 2002; MacDonald et al. 2000]; this level of detail is not needed here. In this paper, we will focus on several characteristics of the generated code and CO_2P_3S that support the theme of this paper: the deferral of design decisions and automated support for design changes.

Using the terminology from [Dietz et al. 2004], the frameworks generated by CO_2P_3S take a *global view* of parallel programming from the perspective of the user. That is, users write their applications without any notion of the multiple processors that will execute it. In a global view of a program that is subdividing an array over several processors, the user would simply write a normal loop iterating over the entire array and the system handles the distribution. The parallel system ensures that each processor works on its own portion of the array. The programmer can concentrate on the application and not on the parallelism. In contrast, a *fragmented view* would require the user to write loops that explicitly iterate over disjoint sections of an array, where each section is assigned to a different processor.

Viewing Template - MorphogenPair

```
/**
 * Prepares this node for the current iteration.
 */
public void prepare( )
{
    this._morphogen1.updateConcentration() ;
    this._morphogen2.updateConcentration() ;
}

/**
 * Iteration method for an interior node in a 4 point mesh.
 */
public void interiorNode( MorphogenPair north, MorphogenPair east, MorphogenPair south, MorphogenPair west )
{
    this._morphogen1.simulate(west.getMorphogen1(),
                              east.getMorphogen1(),
                              north.getMorphogen1(),
                              south.getMorphogen1(),
                              this._morphogen2, 1) ;
    this._morphogen2.simulate(west.getMorphogen2(),
                              east.getMorphogen2(),
                              north.getMorphogen2(),
                              south.getMorphogen2(),
                              this._morphogen1, 2) ;
}

/**
 * Post-iteration processing.
 */
public void postProcess( )
{
    this._morphogen1.updateConcentration() ;
}

/**
 * Gathers results at the end of an entire Mesh computation.
 */
public void reduce( int i, int j, int surfaceWidth, int surfaceHeight, Object reducer )
```

Show Line Numbers | Regenerate After Each Change | Close

Fig. 6: The template viewer in CO_2P_3S.

Editing Code - void

```
public void interiorNode( MorphogenPair north,
         MorphogenPair east,
         MorphogenPair south,
         MorphogenPair west )
{
this._morphogen1.simulate(west.getMorphogen1(),
                          east.getMorphogen1(),
                          north.getMorphogen1(),
                          south.getMorphogen1(),
                          this._morphogen2, 1) ;
    this._morphogen2.simulate(west.getMorphogen2(),
                          east.getMorphogen2(),
                          north.getMorphogen2(),
                          south.getMorphogen2(),
                          this._morphogen1, 2) ;
}
```

OK | Cancel | Restore Default

Fig. 7: The code editor in CO_2P_3S.

It is the responsibility of the programmer to ensure that the sections are disjoint.

In CO_2P_3S-generated frameworks, we achieve this global view by providing hook methods at the level of an individual element in the computation. In the Mesh generative pattern, this corresponds to an individual mesh element or cell in the reaction-diffusion simulation. The framework code uses these elements to construct the larger data structure, populate it with elements, partition the data structure across the processors (using constructor arguments supplied to the framework), and iterate over the partitions. The framework code is also responsible for all of the communication and synchronization that is required for

the parallel mesh computation to work correctly. In other words, the user specifies the behaviour of a single element and the framework builds a parallel mesh program using this behaviour.

The set of exported hook methods includes operation methods for computing new values for mesh elements, control methods for evaluating termination conditions, and additional methods that allow application code to be inserted at various useful points in the Mesh computation. For the reaction-diffusion application, we reused several classes including the class for an individual morphogen. The complete set of hook methods for the Mesh framework are documented in [MacDonald 2002]. Here, we focus on the operation methods as they are impacted by the Mesh generative pattern option values.

The data topology option affects the number of boundary cases that must be considered in a mesh computation. The easiest case is a fully-toroidal mesh. Since all edges wrap around, all elements can use the full stencil to compute new values. For the Mesh framework generated using the above options, the operation method looks like:

```
void interiorNode(MorphogenPair north, MorphogenPair east,
                  MorphogenPair south, MorphogenPair west)
```

This method computes the new value of the current mesh element. The structural framework code invokes this method on every mesh element, using arguments taken from the underlying two-dimensional data structure.

We can now see the three ways in which the operation method is affected by the generative pattern option values. First, the type of the arguments is the mesh element type provided by the user, not a generic type. It is not possible to create a generic interface to the mesh element type because both the data each object contains and the additional helper methods that it may wish to support are application-specific. Second, this reaction-diffusion example uses a four-point stencil, so only four arguments are required. Third, since the simulation uses a fully-toroidal mesh, all mesh elements have all four values. As a result, this single operation method applies to all elements, so it is the only operation hook method provided.

The generative design pattern options correspond to design decisions that must be made during design and development, and their values affect some of the highest-level aspects of the application and framework code. The interface between framework and application code is impacted by the name of the mesh element class, the number of neighbours, and the topology of the data. The structural framework code is also impacted. The synchronization code that is inserted into the framework is determined by the selected synchronization level. Of these options, only the last can be deferred to later in the development process. Initial values for the other options must be selected before coding can begin.

5.3 Changing Design Decisions in the Reaction-Diffusion Computation

The discussion in the previous section outlines the initial development of the reaction-diffusion application. All of the generative pattern options had to be selected before we could start coding the application. During development or possibly during subsequent maintenance some of the characteristics of the application may change, which may mean that the generated framework code is no longer well-suited to the new problem. In this section, we will consider the impact of changes to generative pattern option values. In many cases, the changes are not conceptually difficult. However, they may involve changes

to many source files, which will also need to be retested to ensure that the new application is correct. These are only two of the reasons why design changes late in development are time-consuming and difficult.

In the reaction-diffusion application, there are several changes we can consider:

—We may want to change the mesh element class name to one that better describes its responsibilities.

—We might be interested in generating more complex textures using *anisotropic* simulations, where morphogens diffuse at an angle rather than horizontally and vertically.

—We may decide to use a different data topology. If the texture is large enough then we may not need to tile it, so we can use a non-toroidal topology. If the texture will be mapped onto a cylinder then we only need seamless tiling on one axis, and so we can use a horizontal-toroidal or vertical-toroidal topology.

Each of these changes are interface-affecting. The signatures of the hook methods for application code will change if any of these changes are made, and the structural code must be changed for the new interface. This is why these changes are daunting.

Note that we are not considering a change to the synchronization level option. This option is interface-neutral and can be changed at any time without impact on the user code. However, we have chosen to stay with the original problem specification and thus stay with synchronous iteration.

5.3.1 *Changing the Design Decisions in the Reaction-Diffusion Computation.* Changing the mesh element class seems like a simple change, but it requires that the complete framework be examined for any uses of the old name. However, this change is easily automated using scripts or refactoring facilities in development environments like Eclipse [Object Technology International, Inc. 2006]. This change does impact the interface between the framework and application code (the arguments to the operation method use the mesh element type), but this is a relatively minor problem.

The remaining changes that we will consider affect the reaction-diffusion computation in more substantial ways. These changes also impact the basic, high-level design of the application. Notably, these decisions affect both the structural code and the interface between the structural framework and the application code through the hook methods. As a result, such changes require considerably more developer effort and cannot be automated with facilities in other development environments.

The framework code generated for the reaction-diffusion example used a four-point stencil as the morphogens diffused horizontally and vertically. However, more complex *anisotropic* simulations diffuse the morphogens at an angle. In terms of computation, this change requires that the computation of the new value for an element also include the neighbours on the four diagonals. That is, the four-point stencil is no longer sufficient; we need the eight-point stencil. This change has two consequences.

The first consequence is that the operation method must now include the diagonal elements. The signature must be changed to

```
void interiorNode(MorphogenPair north,
            MorphogenPair northeast, MorphogenPair east,
            MorphogenPair southeast, MorphogenPair south,
            MorphogenPair southwest, MorphogenPair west,
            MorphogenPair northwest)
```

```
public class AbstractRDMeshBlock
{
  public void interiorNode()
  {
    for(int ii = 0; ii < width; ii++) {
      for(int jj = 0; jj < height; jj++)  {
        state.getElement(ii, jj).interiorNode(
                                     state.getElement(ii, jj - 1),
                                     state.getElement(ii + 1, jj),
                                     state.getElement(ii, jj + 1),
                                     state.getElement(ii - 1, jj));
      }
    }
  }
}

public class RDMeshBlockStrategy
{
  // Determines where the partition lies in the global data structure
  // and calls an appropriate method to iterate over the elements in
  // that partition.
  public void operate(AbstractRDMeshBlock meshObj, int x, int y,
                      int meshWidth, int meshHeight)
  {
    // Fully toroidal, so always use interiorNode() with all arguments.
    meshObj.interiorNode() ;
  }
}
```

Fig. 8: Code for enumerating over a fully-toroidal, four-point mesh.

The second consequence is that the portion of the framework code that is responsible for iterating over the elements and calling this method must be changed to call the new operation method with the correct parameters. The second consequence is particularly important as the simulation is a weighted sum of neighbouring element concentration values. The arguments must be supplied in the correct order or the results will be incorrect. Although the changes are not conceptually difficult, they need to be tested.

The differences in the structural code are summarized in the differences between Figures 8 and 9. The top portion of both figures shows the code that is responsible for enumerating over each partition of the data, calling the correct operation method with correct arguments. The eight-point code includes four extra arguments that must be passed in the correct order. The bottom of each figure includes a Strategy class [Gamma et al. 1994] that determines which enumeration method to call based on the location of the partition in the global data surface. Since the code is fully toroidal, the `interiorNode()` is always called, where the `getElement()` accessor method uses modular arithmetic to wrap the edges of the data.

An alternative to supporting multiple stencils is to support a single type of stencil. Individual applications could then base their computations on the relevant subset of supplied neighbouring data. Given a choice between the four-point and eight-point stencil, the eight-point version would be best as it subsumes its four-point counterpart. However, this solution introduces inefficiencies. Passing extra, unnecessary arguments incurs extra costs

```
public class AbstractRDMeshBlock
{
  public void interiorNode()
  {
    for(int ii = 0; ii < width; ii++) {
      for(int jj = 0; jj < height; jj++)  {
        // Need the neighbours on the diagonal, in correct sequence.
        state.getElement(ii, jj).interiorNode(
                             state.getElement( ii, jj - 1 ),
                             state.getElement( ii + 1, jj - 1 ),
                             state.getElement( ii + 1, jj ),
                             state.getElement( ii + 1, jj + 1 ),
                             state.getElement( ii, jj + 1 ),
                             state.getElement( ii - 1, jj + 1 ),
                             state.getElement( ii - 1, jj ),
                             state.getElement( ii - 1, jj - 1 ) );
      }
    }
  }
}

public class RDMeshBlockStrategy
{
  // Determines where the partition lies in the global data structure
  // and calls an appropriate method to iterate over the elements in
  // that partition.
  public void operate(AbstractRDMeshBlock meshObj, int x, int y,
                      int meshWidth, int meshHeight)
  {
    // Fully toroidal, so always use interiorNode() with all arguments.
    meshObj.interiorNode() ;
  }
}
```

Fig. 9: Code for enumerating over a fully-toroidal, eight-point mesh.

as more arguments must be moved to the runtime stack. Though this cost may be small, the cost of accessing the mesh data structure to assemble these arguments may not be. An eight-point stencil generates four extra array memory references which include index computations. As well, since the array is block-distributed among processors, each partition is now adjacent to eight others instead of four. This means that a processor now has to obtain data from eight other processors. An application that only uses the four compass-point neighbours incurs the cost of extra communication for data it does not need, which is clearly undesirable. Thus, explicit support for both four-point and eight-point stencils through code generation is worthwhile.

The last change to the reaction-diffusion computation we will consider is changing the data topology option. The value of this option has the largest impact on the framework structure and its interface with application code. In our reaction-diffusion simulation, we selected a fully-toroidal topology so the texture could be seamlessly tiled. For this topology, only one operation method was required since each mesh element had all four neighbours. If we change the topology, then some elements will be missing neighbours. For a non-toroidal mesh, we have nine boundary cases: four for the corners, four for the edges,

```
void topLeftCorner(MorphogenPair east, MorphogenPair south)
void topEdge(MorphogenPair east, MorphogenPair south,
                  MorphogenPair west)
void topRightCorner(MorphogenPair south, MorphogenPair west)
void leftEdge(MorphogenPair north, MorphogenPair east,
                  MorphogenPair south)
void interiorNode(MorphogenPair north, MorphogenPair east,
                  MorphogenPair south, MorphogenPair west)
void rightEdge(MorphogenPair north, MorphogenPair south,
                  MorphogenPair west)
void bottomLeftCorner(MorphogenPair north, MorphogenPair east)
void bottomEdge(MorphogenPair north, MorphogenPair east,
                  MorphogenPair west)
void bottomRightCorner(MorphogenPair north, MorphogenPair west)
```

Fig. 10: Signatures for the operation methods for a four-point mesh.

and one for the interior. Horizontal-toroidal and vertical-toroidal topologies have three boundary cases: two for the edges that do not wrap and one for the interior.

Changing the topology requires changes to the structural code that iterates over the mesh elements assigned to each processor. The location of each element must be checked so that the proper set of neighbouring elements can be determined. Further, in CO_2P_3S, the generated framework code includes a separate operation method for each boundary case, affecting the interface to the hook methods that make up a user application. The structural code determines which method is appropriate and calls it with the correct neighbour arguments. The set of nine possible operation methods for a four-point stencil are shown in Figure 10. The methods for an eight-point stencil also include arguments for the available diagonal elements. CO_2P_3S takes the additional step of introducing only the subset of operation methods that are applicable for the selected topology. Otherwise, the user will see operation methods that are never invoked, which would be confusing.

The necessary changes in the framework structure are summarized in the differences between Figures 11 and 8. The Strategy class at the bottom of the figures must now choose a method to enumerate over a given partition based on its location. Each partition will have a set of boundary conditions to consider, and must invoke the appropriate operation method, as shown in the top of the figures. The top part of Figure 11 shows one example of a new partition enumeration method for the data at the top left corner. Data at the corner and edges must be handled separately, before the data in the middle of the partition is computed using the `executeInteriorNodes()` method. There are a total of 16 different partition enumeration methods included in the code (the nine shown in the Strategy class, and six more to address cases where the number of vertical or horizontal partitions is one and a partition must account for multiple edges in a single partition).

Again, while this change is conceptually simple, in practice it is tedious, time-consuming, and error-prone. The operation method that should be invoked and the order of its arguments are crucial for the correctness of the code. Further, the effects of the stencil and topology are not independent of each other. Changing the number of neighbours in a non-toroidal mesh requires more effort than the same change in a fully-toroidal mesh since the number of operation methods is larger.

With CO_2P_3S, the user can update not just the design but also the implementation of the application structure in just a few minutes, not including the time for application code

```
public class AbstractRDMeshBlock
{
  // Example iteration method, to call operation methods for elements
  // for a partition. There are 16 methods for non-toroidal meshes.
  // In this example, iterate over the elements in a partition that is
  // in the top left corner, with edges on top and to the left.
  public void topLeftCorner( )
  {
    state.getElement(0, 0).topLeftCorner(state.getElement(1, 0),
                                         state.getElement(0, 1));
    for(int j = 1;j < height;++j) {
      state.getElement(0, j).leftEdge(state.getElement(0, j - 1),
                                      state.getElement(1, j),
                                      state.getElement(0, j + 1));
    }
    for(int i = 1;i < width;++i) {
      state.getElement(i, 0).topEdge(state.getElement(i + 1, 0),
                                     state.getElement(i, 1),
                                     state.getElement(i - 1, 0));
    }
    executeInteriorNodes(state, 1, width, 1, height) ;
  }
}

public class RDMeshBlockStrategy
{
  // Determines where the partition lies in the global data structure
  // and call an appropriate method to iterate over its elements.
  public void operate(AbstractRDMeshBlock meshObj, int x, int y,
                      int meshWidth, int meshHeight)
  {
    if( x > 0 && x < meshWidth - 1 && y > 0 && y < meshHeight - 1 ) {
      meshObj.interiorNode();
    } else if( x == 0 && y == 0 ) {
      meshObj.topLeftCorner();
    } else if( x == 0 && y == meshHeight - 1 ) {
      meshObj.bottomLeftCorner();
    } else if( x == meshWidth - 1 && y == 0 ) {
      meshObj.topRightCorner();
    } else if( x == meshWidth - 1 && y == meshHeight - 1 ) {
      meshObj.bottomRightCorner();
    } else if( x == 0 ) {
      meshObj.leftEdge();
    } else if( x == meshWidth - 1 ) {
      meshObj.rightEdge();
    } else if( y == 0 ) {
      meshObj.topEdge();
    } else if( y == meshHeight - 1 ) {
      meshObj.bottomEdge();
    }
    // Additional cases where number of vertical or horizontal
    // partitions is only one, so a partition has multiple edges.
  }
}
```

Fig. 11: Simplified code for enumerating over a non-toroidal four-point mesh.

Table II: Number of statements for different frameworks generated from the Mesh generative pattern.

Architecture	Fully-toroidal	Non-toroidal	Horizontally-toroidal	Vertically-toroidal
Shared Memory	295	695	410	410
Distributed Memory	962	1870	1229	1232

(a) Number of statements in various four-point mesh frameworks.

Architecture	Fully-toroidal	Non-toroidal	Horizontally-toroidal	Vertically-toroidal
Shared Memory	315	794	458	458
Distributed Memory	1080	2259	1424	1432

(b) Number of statements in various eight-point mesh frameworks.

changes. The new option values are entered using the CO_2P_3S user interface and the code regenerated. Since these changes are interface-affecting, both structural and hook method classes must be regenerated. Changing the stencil and topology is a matter of changing the Mesh generative pattern option values and regenerating the code. This regeneration produces a new framework for the new option values. For hook methods that are identical in both the previous and regenerated code, application code is automatically re-inserted into the new hook method class. There may also be new hook methods that the user has to implement, and some hook methods may disappear. It is also possible that some of the methods have different signatures.

Regardless of the changes to the interface between the structural framework code and the application-specific hook method code, it is also crucial to note that the structural framework code has also been updated with a new implementation based on the generative pattern option values. This new framework obtains the necessary data from other processors, correctly determines which hook methods to call, and provides the proper arguments in the right order. The user only has to augment or modify the hook method code, knowing that the structure will use these methods to execute their application.

To provide an idea on the scope of the changes, without any application code the multithreaded four-point fully-toroidal mesh framework generated by CO_2P_3S consists of 295 statements of Java code. The multithreaded eight-point non-toroidal mesh takes 794 statements, over 2.5 times as much code. With a few values entered using the GUI, CO_2P_3S can introduce 499 lines of correct multithreaded Java statements into an existing parallel application. The necessary changes affect three of the seven classes generated for the Mesh generative pattern. Once the application-specific hook method code is adapted to the new design, the program runs correctly.

The number of statements for all variations of the framework generated for the Mesh generative pattern, for different option values and execution architectures, is given in Table II. We measured code size in terms of statements, as measured by the Metrics Plug-in for Eclipse [Team in a Box Ltd. 2002], to remove any effects from code formatting that could artificially inflate or deflate these numbers. Also note that these counts include some optimizations, which are discussed in Section 5.4.2.

5.4 The Impact of the Execution Architecture

One important aspect of a parallel program that was not part of the Mesh generative design pattern is the execution architecture for the resulting program. The previous section described changes to a multithreaded implementation of the reaction-diffusion program. The generative patterns also include an option that allows the user to select between shared-memory and distributed-memory environments.[2] This option has a large impact on the structural code, for reasons that we will make clear in the next section.

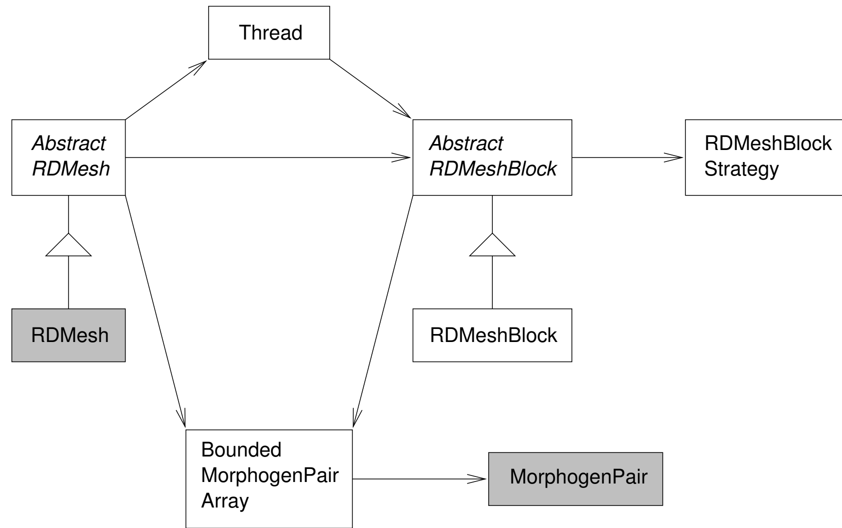
The effects on code size are clearly demonstrated in Table II; moving from a shared-memory to a distributed-memory framework almost triples the size of the generated code. The distributed framework also includes several classes to deal with issues like exchanging of border information and registering objects with a distributed name service. The shared-memory and distributed-memory frameworks are presented in Figure 12 for comparison.

However, changing the execution architecture is interface-neutral. Only the structural code is affected; there is little impact on either the interface of the hook methods or the application code inside. This is the result of a global view of the computation. The user focuses on the computation, which is independent of the execution architecture. This independence is further reinforced by using an object-oriented framework to separate parallel structural code from application code. The application code is not polluted with any architectural details, such as the creation and management of concurrency, data distribution, synchronization, and communication. These aspects are all part of the design pattern on which the generative pattern is built, so the generated code addresses these problems.

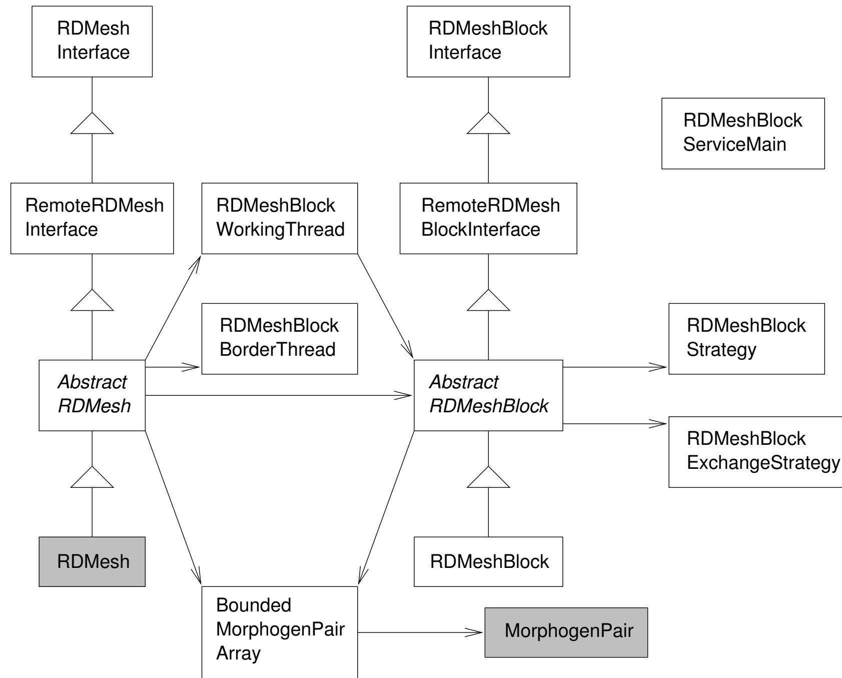
Because the choice of execution architecture is an interface-neutral option, a value need not be selected until just before the structural code of the framework is generated. In the current version of CO_2P_3S, this generation process is done much earlier, after all option values for a generative pattern are supplied. That is, CO_2P_3S always generates a complete framework for the generative pattern, including both structural classes and hook method classes, at this time. However, it is always possible to change option values and regenerate the code at any time. Another approach is to split the generated code into two parts, structural classes and hook method classes. The hook method classes would be generated after the interface-affecting option values have been specified, as these are needed before coding can begin. Structural framework code could be generated later (during the compilation process, for example). Thus, the choice of values for interface-neutral options could be deferred until just before the structural code is created. Changes to these options would only require a recompilation. This is possible since the hook method code is largely independent of the structural code. It would also have the side effect of improving the responsiveness of the tool when code was generated during development since there would be much less code to produce.

The next section briefly details the differences between shared-memory and distributed-memory architectures from a programming perspective. The following section describes the process of porting the reaction-diffusion application from the multithreaded version in the previous section to a distributed-memory architecture.

[2]The current implementation of CO_2P_3S has separate generative patterns for shared-memory and distributed-memory versions, to simplify development. Using the MetaCO_2P_3S system for creating and modifying generative design patterns in conjunction with our code generation system [Bromling 2001], the two generative patterns could be combined into one with an option for the execution architecture. Thus, for this paper, we will assume that execution architecture is a generative pattern option.



(a) Shared-memory framework.



(b) Distributed-memory framework.

Fig. 12: Shared-memory and distributed-memory frameworks for the Mesh generative pattern. The classes in grey boxes are exposed to users at the Patterns Layer in CO_2P_3S.

5.4.1 *Shared-Memory versus Distributed-Memory Parallel Programming.* There are two main architectures for parallel programming. The *shared-memory* architecture consists of a set of processors that are physically or logically connected to a single, common memory. Different processes or threads execute on different physical processors and can use the common memory to exchange data. The *distributed-memory* architecture consists of a set of processors with their own private memories, connected using a network. Since memory is private to a processor, data must be exchanged by explicitly sending messages across the network.

One of the difficulties in parallel programming is that the choice of architecture affects the application code. Parallel application code, when not written with the benefit of a high-level tool, includes not only the computation but also the code that manages the parallelism and hardware resources used to solve the problem. This extra code is a function of the selected architecture and is not a function of the problem being solved. Time spent on this code is time not spent on the problem.

Worse, the code that manages and controls the parallelism depends on the choice of architecture. Distributed-memory programs must include explicit message passing primitives to exchange data where shared-memory programs need only set values at common addresses. Shared-memory programs usually include more synchronization primitives to control the order of events in the system where distributed-memory programs achieve this synchronization through blocking message passing. Regardless, these parallel primitives tend to be scattered throughout the application code and become an integral part of the program.

Adding to the problem is that the different architectures usually have different execution characteristics that must be considered in the design and implementation. The relative costs of operations, particularly memory accesses or message transmissions, can be orders of magnitude different. Shared-memory architectures have a dedicated high-speed internal interconnection network connecting the processors to memory that allows for fast access. Distributed-memory architectures may rely on off-the-shelf networking technologies like Ethernet or faster gigabit networks. The cost of communication in the distributed-memory case is much higher as operating system calls and network protocol overheads must be factored in. This long latency requires extra design so that it does not limit performance, usually by overlapping communication with computation. Though communication is much faster in the shared-memory case, it must still be considered to achieve the best possible performance. This is usually addressed by carefully considering data locality.

The result of these problems is that application developers generally choose their target architecture early in the design stage so that the necessary architectural issues can be carefully considered. Once the application has been developed for a particular architecture, it takes substantial effort to port to a new architecture. In fact, even changing the communication structure of an application within the same architecture is often prohibitively difficult.

It is possible to run a program written for one architecture on the other given appropriate software, though usually with reduced performance. A distributed-memory program can be run on a shared-memory multiprocessor by running multiple processes on the machine and allowing them to communication through message passing, though this still incurs OS and protocol overheads. The message passing primitives could be rewritten to deposit messages in a bounded buffer to avoid this overhead, but this incurs more data copying than might be necessary in a shared-memory implementation. A shared-memory program can

be run on distributed-memory hardware using software abstractions like distributed shared memory (such as TreadMarks [Amza et al. 1996]). However, this software does not reduce the long latency for accessing remote data so locality is even more crucial to performance. This problem can be reduced by carefully considering the memory consistency model used in an application [Lu 2000].

5.4.2 *Porting Reaction-Diffusion from Shared-Memory to Distributed-Memory Architecture.* When porting a CO_2P_3S program from shared-memory to distributed-memory, it is important to point out that the application code in the hook methods may require a small number of changes but the interface to the hook methods is unchanged. Again, this result is a combination of our global view of the computation and our use of object-oriented frameworks to separate application code from parallel structural code. Both shared-memory and distributed-memory versions of a framework generated using a generative design pattern export the same set of hook methods. In fact, the changes that were necessary for the reaction-diffusion application are the result of limitations in Java RMI and serialization.

The first difference between the two versions of the application is that the mesh element class must be serializable; that is, the Java virtual machine must be able to flatten the object into a stream suitable for network transmission. Many classes can do this using the serialization facilities built into Java, which only require that the class implement the `Serializable` interface and follow a small number of code conventions. These conventions are that the class provide a default, no-argument constructor for the class (for use during object reconstruction) and that any instance variables also be serializable. The latter convention can be avoided by having the user write their own serialization code, though we use the provided serialization routines. We can simply add the `Serializable` interface to the mesh element class during compilation for the distributed-memory environment.

One problem with the port was a limitation in that Java serialization does not include the values of static variables for the class. In the reaction-diffusion code, static variables were used to hold simulation constants to reduce the size of a mesh element object. These variables were initialized in the mainline for the program. To port the reaction-diffusion application to distributed-memory, the static variables had to be initialized in the constructor of the mesh element class. Each processor sets these variables for every mesh element in its local partition, which resulted in more assignments than were necessary, but this was the simplest solution. Unfortunately, this must be done in the application-specific code and cannot be automated. However, this change will not prevent this code from running in the shared-memory environment.

The most important difference between the two versions of the reaction-diffusion application lies in the structural code. From the previous section, we know that there are significant differences in distributed-memory programming. In the reaction-diffusion example, there are three key differences:

(1) The distributed-memory version requires the explicit exchange of data between processors responsible for adjacent partitions of the mesh data. In contrast, the shared-memory version must include synchronization to control access to shared data and control the timing of loops assigned to different threads.

(2) The distributed-memory version physically partitions the data into disjoint partitions that are sent to different processors, where the shared-memory version logically partitions a single copy of the data.

(3) The relative cost of communication is much higher in the distributed-memory version as the network and protocol overheads are more expensive.

The first two differences primarily affect the structure of the generated framework code. They dictate the primitives that are needed to correctly implement the parallel structure and how they are used. The last difference, the relative costs of communication between the two architectures, raises another interesting problem. The communication costs determine the set of optimizations that can be meaningfully applied in the Mesh framework code.

The CO_2P_3S system automatically includes several optimizations into the generated framework code. These optimizations are conservative in that they do not affect the correctness of the application code provided it follows the documented framework assumptions. If more aggressive optimizations are possible or if the implemented ones do not improve performance, CO_2P_3S users have the ability to change the edited code using the Intermediate and Native Code layers.

Applying such optimizations makes the Mesh framework larger and more complex. This helps demonstrate the benefits of a pattern-based programming system. These optimizations are automatically applied and inserted by the programming tool, and the user benefits from them with no additional effort. Further, they are automatically applied when the generative pattern options change. In contrast, without tool support, the user would be forced to write the extra code for these optimizations by hand and verify its correctness, and this extra code would need to be properly maintained if the implementation changed.

To highlight these differences and show the impact that the execution environment can have on the implementation of the reaction-diffusion application, we will consider two optimizations that can be applied. The first optimization reduces the use of modular arithmetic to compute array indices for accessing mesh data, and can be applied to both shared-memory and distributed-memory architectures. The second optimization is to overlap computation with communication, which is only applicable in the distributed-memory version of the reaction-diffusion program.

5.4.2.1 *Reducing the Use of Modular Arithmetic.* The first optimization, common to both architectures, is to reduce the amount of modular arithmetic used in the calculation of array indices for accessing mesh data. The easiest method of ensuring that the array indices do not go out of bounds when using a topology that wraps any edge is to use the index modulo the data size. However, simple tests suggest that modular arithmetic is between 1.25 and 8 times slower than its non-modular counterpart, depending on the modulus value. Modulus values that are powers of 2 are less expensive. Additional care must be taken when dealing with negative numbers in modular arithmetic since they return negative values that cannot be used as array indices, which can further increase the cost. Given the large number of array indices that are computed in a mesh computation, minimizing the use of modular arithmetic is a worthwhile optimization.

In the distributed-memory case, the use of modular arithmetic can be avoided altogether by using *ghost boundaries*, illustrated in Figure 13. The size of the 2D data structure allocated at a processor is larger than the size of the local partition. The local partition forms the center of the data structure, and the extra rows and columns are used to hold data from adjacent partitions sent to the processor via message passing. The processor only computes new values for the interior. With ghost boundaries around all sides, it is not possible for the array index value to be out of bounds, making modular arithmetic unnecessary.

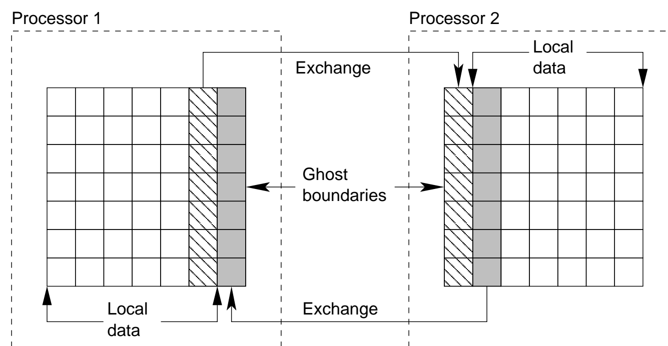


Fig. 13: The exchange of ghost boundaries in distributed-memory meshes.

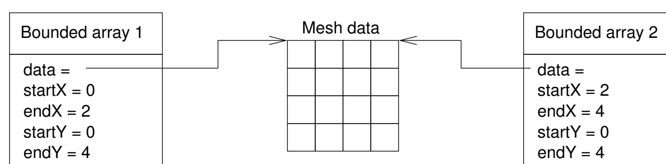


Fig. 14: A bounded array for shared-memory meshes.

In the shared-memory case, we can take the same approach of physically partitioning data for processors. However, this incurs copying overhead that is unnecessary. An alternative is to use a bounded array, shown in Figure 14. We create a bounded array object whose bounds cover a subarray of the original, larger data set. We can create many of these bounded arrays, each sharing a single copy of the data. Array indices are translated by accessor methods in the bounded array class. This approach conserves memory in the shared-memory version of the program.

The benefit to using a bounded array in shared-memory is that we also no longer need ghost boundaries. When computing new values, a thread can access array elements from adjacent partitions by simply accessing the appropriate element outside of its local partition, with sufficient synchronization. We assume that each mesh element contains both a read value of its data for computation and a write value for update to avoid data races (which is reasonable since a mesh element must also determine when it has converged to its final value by comparing results from successive iterations).

The need for modular arithmetic with bounded arrays depends on how much information the object has about the size of the original data and the location of its bounds within the larger data set. Our bounded arrays hold no such information, so we use modular arithmetic when accessing neighbouring elements on the edge of a bounded array. This may not be necessary (as the bounded array may be defined completely within the larger data structure), but is conservatively safe. Modular arithmetic is not needed when processing interior elements. As a result, we split the iteration of a partition across several loops, one for each edge condition and one for the interior.[3] Only the former loops use modular arithmetic. The latter (and usually larger) loops use normal arithmetic. This is shown in Figure 15, which is the code from Figure 8 rewritten to reduce the use of modular

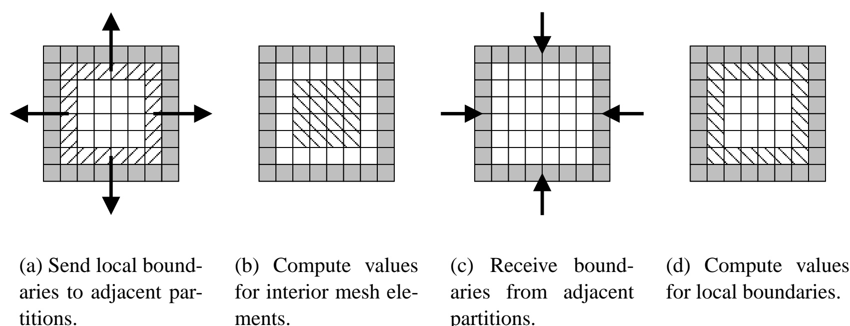
[3]The framework documentation notes that the user cannot assume the iteration over a partition happens in any particular order.

```
public class AbstractRDMeshBlock
  public void interiorNode( )
  {
    // Process elements on the edges using modular arithmetic as they
    // may wrap around to the other edge if the current partition
    // contains an edge.
    for(int jj = 0; jj < height; jj++) {
      state.getElement( 0, jj ).interiorNode(
                  state.getElementUsingModulus( 0, jj - 1 ),
                  state.getElementUsingModulus( 1, jj ),
                  state.getElementUsingModulus( 0, jj + 1 ),
                  state.getElementUsingModulus( -1, jj )) ;
      state.getElement( width - 1, jj ).interiorNode(
                  state.getElementUsingModulus( width - 1, jj - 1 ),
                  state.getElementUsingModulus( width, jj ),
                  state.getElementUsingModulus( width - 1, jj + 1 ),
                  state.getElementUsingModulus( width - 2, jj )) ;
    }
    for(int ii = 1; ii < width - 1; ii++) {
      state.getElement( ii, 0 ).interiorNode(
                  state.getElementUsingModulus( ii, -1 ),
                  state.getElementUsingModulus( ii + 1, 0 ),
                  state.getElementUsingModulus( ii, 1 ),
                  state.getElementUsingModulus( ii - 1, 0 )) ;
      state.getElement( ii, height - 1 ).interiorNode(
                  state.getElementUsingModulus( ii, height - 2 ),
                  state.getElementUsingModulus( ii + 1, height - 1 ),
                  state.getElementUsingModulus( ii, height ),
                  state.getElementUsingModulus( ii - 1, height - 1 )) ;
    }

    // Process interior nodes without modular arithmetic.
    executeInteriorNodes( state, 1, width - 1, 1, height - 1 ) ;
  }
  protected void executeInteriorNodes(int startX, int endX,
                                      int startY, int endY)
  {
    // getElement does not use modular arithmetic since it is
    // only used on interior elements.
    for(int ii = startX; ii < endX; ii++) {
      for(int jj = startY; jj < endY; jj++) {
        state.getElement( ii, jj ).interiorNode(
                state.getElement( ii, jj - 1 ),
                state.getElement( ii + 1, jj ),
                state.getElement( ii, jj + 1 ),
                state.getElement( ii - 1, jj ) ) ;
      }
    }
  }
}
```

Fig. 15: The code from Figure 8 rewritten to reduce modular arithmetic.

arithmetic by unrolling the loops over a given partition.



(a) Send local boundaries to adjacent partitions.

(b) Compute values for interior mesh elements.

(c) Receive boundaries from adjacent partitions.

(d) Compute values for local boundaries.

Fig. 16: Overlapping communication and computation in a mesh. The grey cells are the ghost boundary.

5.4.2.2 *Overlapping Communication and Computation.* The second optimization is applicable to distributed-memory mesh applications. In this architecture, communication costs are higher and, if treated naïvely, can create serious performance problems. One way to hide the communication costs is to overlap the communication with computation. That is, a processor should initiate communication but rather than immediately waiting for a reply, it should execute some useful work first. If the computation is large enough, the reply will have arrived and can be read without blocking. If only synchronous communication is available (like Java RMI), the communication can be done by separate threads. Similar problems can occur in shared-memory computations on non-uniform memory architectures, but the technique described here is not applicable to those situations because of the high cost of threads compared to memory accesses.

In a mesh computation, this can be applied to the exchange of boundary elements between adjacent partitions. This is shown in Figure 16. A processor sends its boundary elements to the processors holding adjacent partitions. It then computes new values for elements in the interior of its local partition, which uses only local data. After this is complete, the processor then receives its new boundaries from other adjacent partitions, and uses these values to compute new values for elements on the edge of its partition.

5.4.2.3 *The Effects of Optimization.* An important characteristic of both of these optimizations is that they result in more structural code. In both cases, the computation is broken into several loops for different boundary cases to achieve its goal. Although this improves the performance of the application, it can make changing other design decisions more difficult. Changing the stencil means that each loop must be changed to access new neighbouring data elements. Changing the topology may increase the number of boundary cases that must be considered, requiring new loops in the program. These new cases will require more code if the optimizations are to be maintained. In both changes, the distributed memory version may also require more communication to access new neighbouring data elements.

With a pattern-based programming system, these optimizations can be applied automatically when structural code is generated. More importantly, as generative pattern option values change, the optimizations are automatically applied to new code that is introduced. As a result, application maintenance is no longer hampered by the inclusion of this optimized code.

Adding to this problem is that not only can the chosen architecture influence the code,

but the particular parallel computer system can also be a factor. Many systems now fall into the category of non-uniform memory access or NUMA systems. In these systems, each processor provides a subset of the total global address space. The processors can access any memory system, but the relative cost of access depends on where the memory resides. Accessing local memory is relatively inexpensive, but remote memory is more expensive. Recalling that memory access costs was one of the differences between distributed-memory and shared-memory architectures, some optimization techniques for the former architecture may be useful in the latter. This work has not considered optimizations for individual parallel computer systems, though this would certainly be possible given our conditional code generation process.

5.5 Limitations to Design Decision Changes

The CO_2P_3S system allows different options for a given generative design pattern to be changed easily. New structural framework code can be generated easily. Depending on the changes, there may be new hook methods that must be implemented. However, code from the previous version will be incorporated into the new hook methods where possible.

Although not needed in the reaction-diffusion example, it is possible to compose the generated frameworks. More complex programs may require different parallelization strategies for different computational phases. In a CO_2P_3S program, this composition is accomplished by instantiating and using another generated framework in the hook method code for another framework. This composition has been used in the implementation of a parallel sorting application [MacDonald et al. 1999; MacDonald 2002].

The largest limitation of the CO_2P_3S system with respect to altering design decisions for generative patterns is that it is difficult to change the generative pattern applied to a problem. If the user wishes to experiment with alternative parallelization strategies, the hook method code from one framework is unlikely to be easily reused in hook methods for another framework. This problem can be mitigated using delegation. Rather than implementing the problem in the hook methods, a better strategy is to delegate hook method code to another object that holds the problem implementation. Although the hook method code will still not be reusable if the generative pattern changes, the amount of code that must be rewritten is small.

5.6 The Need for Extensibility

Although we have concentrated on automated support for changing design decisions, one of the unique qualities about CO_2P_3S is its support for openness, in the sense that the structural code is available for the user to modify at the Intermediate Code and Native Code layers. At these layers, if the code has been modified, changing it incurs all of the problems that we are striving to avoid. Thus, we need to minimize the need for the user to work with the structure at these lower layers.

A related problem that affects all design-pattern-based systems is that it is impossible to foresee the complete set of design patterns that a system should support. It is difficult to predict the needs of all users, and new design patterns are constantly being created. If a needed pattern does not exist, then users are forced to either wait for the system developer to provide it or abandon the tool.

The CO_2P_3S system deals with this problem by providing the MetaCO_2P_3S facility, which is described in more detail in [Bromling 2001; Bromling et al. 2002; MacDonald et al. 2002; Anvik et al. 2002]. This facility provides tool support for creating new

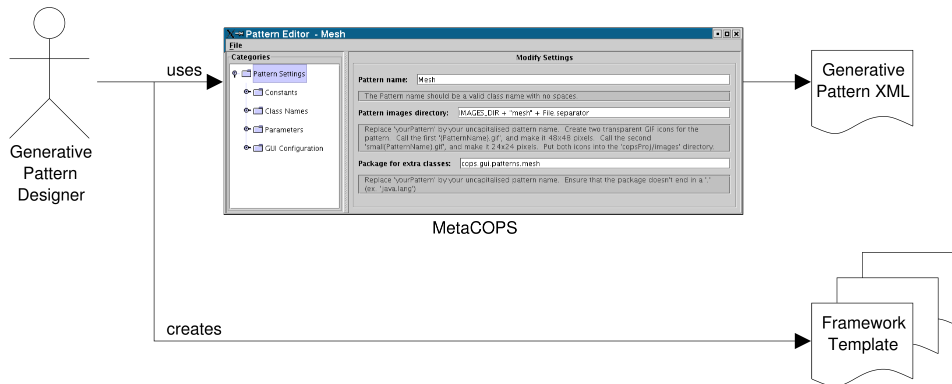


Fig. 17: A brief overview of the MetaCO_2P_3S system.

generative patterns or modifying existing ones. Modifications to a generative pattern can include adding a new option and changing the generated structural code based on its value. These modified patterns can be incorporated into the CO_2P_3S system and used exactly as the generative patterns supplied with the tool. In fact, all of the generative patterns in CO_2P_3S were created using MetaCO_2P_3S.

A brief overview of MetaCO_2P_3S is shown in Figure 17. A generative pattern designer uses the MetaCO_2P_3S tool to specify details about the generative design pattern, including any options that the CO_2P_3S interface must support. Once this is finished, the designer then writes a framework template for the code generator. This framework template uses a combination of JavaDoc tags and text substitution to specify how the generated code changes based on values of the pattern options.

In CO_2P_3S, the framework template and option values are processed by a code generator based on JavaDoc [Pollack 2000]. JavaDoc tags are used to conditionally include certain methods, and text substitution is used to replace class names. In addition, code fragments within a method body can be conditionally inserted based on option values. This code generator could be replaced with newer template-based code generation systems like Velocity [Apache Jakarta Project 2005], which were not available when MetaCO_2P_3S was under development.

Once complete, the generative pattern XML and framework template are incorporated into the CO_2P_3S user interface by importing the pattern. Once imported, the generative pattern appears in the pattern palette on the left-hand side of the CO_2P_3S interface (in Figure 4), and are used as shown in the CO_2P_3S overview (in Figure 2).

If users discover commonly recurring changes that they make at lower layers in CO_2P_3S, then the generative pattern can be changed to generate this code directly. Once this is done, the benefits of decision deferral and automated support for changes is once again possible.

6. PERFORMANCE

The most important consideration in the parallel programming domain is application performance. We present performance results, in the form of speedup, for the reaction-diffusion application in Tables III and IV. For both programs, the results do not include the time to distribute data across processors or gather results; only the time to compute the texture is included. The speedups were computed based on a separate, sequential implementation of the problem.

Table III: Performance of the shared-memory reaction-diffusion application using four processors.

Mesh Size	Speedup CO_2P_3S code
800 × 800	3.5
1200 × 1200	3.5

Table IV: Performance of the distributed-memory reaction-diffusion application using four processors.

Mesh Size	Speedup CO_2P_3S code	Speedup optimized code
800 × 800	1.1	1.7
1200 × 1200	2.0	2.9

The results for the distributed memory version of the reaction-diffusion application include two versions of the program. The first version of the code is for the code generated for the distributed-memory mesh supplied with CO_2P_3S. With four processors, the smaller problem size runs only 10% faster than the sequential code, where the larger size is twice as fast. Although these initial results were disappointing, there are three important points to consider. First, the distributed-memory version can be created from the shared-memory version with almost no changes to the application code. The only changes to the simulation code are those dealing with the static variables for the simulation constants, which required the changes described in Section 5.4.2. Otherwise, the simulation code can be reused directly from the sequential version. The hook method bodies for the generated framework are unchanged from those in the shared-memory code. Second, the changes to the structural code needed to port the program from the shared-memory version to the distributed-memory version are handled by the code generation system in CO_2P_3S. The execution architecture is normally a fundamental design decision that must be made almost immediately in the design process as it impacts the rest of the application design and implementation. Changing this decision after the application has been implemented would be a major undertaking. With CO_2P_3S, most of the changes are accomplished in minutes. Third, while the performance may be disappointing, the results are correct. Any optimization efforts will start with a correct program. Further, if optimization efforts introduce errors, the correct version can be regenerated at any time.

In [Tan et al. 2003], it was observed that the reason for the somewhat disappointing results was high synchronization overhead, introduced by the use of two distributed barriers in each mesh iteration. Although needed in general, it was possible to move some of the application code and remove one of the barriers in this application. One of the benefits to the PDP process and the CO_2P_3S implementation is that the generated framework code is presented to the user so such optimizations can be applied. In many other programming systems, the closed nature of the tool would prevent such changes. This change, applied manually by the application developer, improved the performance of the application as shown in the optimized column in Table IV.

Regardless, the performance of the distributed-memory version also improves as the problem size increases. As the problem grows, the computation outweighs the costs of message passing and synchronization, yielding better speedups.

The shared-memory version of the program obtains good results for four processors for both problem sizes without the need for additional optimizations.

For all CO_2P_3S programs, the performance results may be less than those that could be obtained through a hand-coded implementation with low-level threading or message-

passing libraries. However, we feel that the decrease in development time often makes up for the performance losses. This idea is the impetus behind the High Productivity Computing Systems initiative sponsored by DARPA [HighProductivity.org 2005]. In situations where high performance is crucial, the developer has the option of using the lower layers of CO_2P_3S to access the implementation of the generated code to correct performance problems, as was done with the distributed reaction-diffusion application. If performance is still unacceptable, the developer can write a customized application, using the CO_2P_3S version of the program as an initial prototype to determine the most effective parallel structure for the application.

7. RELATED WORK

This section examines related work. First, we examine some research in maintaining information about the design decisions that make up a software system. Second, we focus on programming systems supporting the implementation of design patterns, in both the parallel programming and sequential programming domains. For these systems, we try to differentiate between different systems based on how they allow users to change existing programs. However, most of these systems only describe initial development stages, so some extrapolation is necessary. Finally, we examine research into the use of aspect-oriented programming [Kiczales et al. 1997] in the implementation of design patterns.

7.1 Design Decision Tools

A serious problem in software engineering is *design erosion*, where the design of a software system degrades over time and maintenance becomes increasingly difficult [van Gurp and Bosch 2002]. Eventually, systems degrade to the point where developers feel it is best to redesign the complete system rather than maintain the existing one. One source of design erosion is an inability to relate source code to the underlying design decision. Some effort has been made to maintain this relationship, as well as document the rationale for design decisions for the benefit of future maintainers, representing design decisions as first-class entities in a software project [Jansen and Bosch 2004; 2005].

In this research, we only consider design decisions related to the generative design patterns selected by the user to solve their problem and not the more general problem. However, in limiting the problem scope, we are able to provide better support for the use of design patterns in application programming.

7.2 Pattern-based Programming Tools and Systems

7.2.1 *General Pattern-based Tools.* There have been many research efforts to support design patterns in the development process for sequential programs. This section describes a representative sample of this work.

Budinsky *et al.* created a web-based system for generating C++ implementations of design patterns [Budinsky et al. 1996], supporting the patterns described in [Gamma et al. 1994]. Like the generative patterns in CO_2P_3S, each pattern includes some structural options, taken from the "Implementation" section of [Gamma et al. 1994]. The generated code includes the physically-common code and a small set of the logically-common code, specifically the logically-common code that does not involve application-specific interfaces. For example, in the Composite generated by this tool, the user can provide the names of subclasses for both composite and leaf classes and can select which of these classes support the child management methods (just the composites or all classes) since

these only require user-supplied class names. Basic C++ code for each class is generated if desired (constructors, copy constructors, virtual destructors) and the child management code is inserted into the appropriate classes. No application-specific methods can be specified. The system will generate an example use of each design pattern, but the code is artificial and cannot be controlled by the user, meaning it can only be used as a guideline for the correct implementation of application-specific methods. It would be possible to use new option values to generate new code that implements application-independent concerns (such as moving the child management methods in the Composite), but any application-specific code would need to be copied to the new classes. Further, the user would be responsible for any changes to any code involving application-specific code. In contrast, CO_2P_3S generates code more specific its eventual use, which can be changed with less user intervention.

The tool described in [Florijn et al. 1997] represents a pattern as a graph of *fragments*. Fragments are Smalltalk objects representing various design elements (classes, methods, instance variables) and relationships between them (association, inheritance). These fragments can be used in two ways. First, existing code can be identified as playing a part in a design pattern by assigning different elements to the corresponding fragment, to help document patterns in the system. Second, fragments can be assembled into a description of a new pattern instance, which can be used to generate code for the new pattern. In both uses, the fragment description can be checked for conformance to a pattern specification to ensure pattern constraints are met. Further, refactoring operations can be applied to a pattern instance to reorganize the code without violating the pattern specification. However, the member of the pattern family that is supported is determined by the *root fragment* that describes the pattern structure. It is not clear that these roots can produce or recognize pattern variations; it is possible that each pattern variation is represented by a new root fragment, which incurs the problems noted in Section 3. This may also make it difficult to change an existing pattern as fragments may need to be copied to a new root fragment to instantiate a pattern variation. Later work used fragments to refactor code using a simple web browser as an example [van Winsen 1996], but the small number of supported refactorings limited the experience. CO_2P_3S does not help document existing patterns, but only allows pattern code to be generated. However, the generative patterns in CO_2P_3S are higher-level and easier to instantiate than assembling a (possibly large) number of individual fragment objects.

FRED (and its followup JavaFrames) [Hammouda and Koskimies 2002; Viljamaa and Viljamaa 2002] and MADE [Hammouda et al. 2004; Siikarla and Systä 2006] provide a general programming environment that supports a task-driven methodology for specializing an object-oriented framework. These projects view a pattern implementation as a set of *roles* and *constraints*. A role is a participant in a relationship defined by a design pattern. In order to properly play a role, a class must properly implement a set of properties. Each property is a constraint of the role. FRED uses these roles and constraints to guide the developer through the process of implementing a pattern. This is done by presenting a list of outstanding tasks that must be finished before the pattern implementation is complete, using a facility similar to the task view in Eclipse [Object Technology International, Inc. 2006]. Where possible, code generation facilities can automatically generate a default implementation of this code, which can be customized by the user, and insert it into the application. Constraints can be validated by the tool, and the user can be asked to correct any errors. CO_2P_3S does not bind pattern roles to existing code, which allows it to gener-

ate a pattern implementation in terms of exposed hook methods. Further, CO_2P_3S is not supporting an existing framework but generates its own.

JavaFrames was extended to support maintenance patterns to guide users through maintenance tasks in a software system [Hammouda and Harsu 2004]. A pattern is once again represented as a series of roles and constraints. However, the patterns are no longer framework specialization but rather capture relationships between program elements that are needed to accomplish maintenance tasks. If an element is changed, the maintenance pattern produces a task list of further changes needed to complete the maintenance task. Again, where possible, the needed changes can be automated through code generation. This requires the developer to identify the maintenance patterns in the application and associate the patterns with the correct program elements. At the Patterns Layer, CO_2P_3S hides a pattern implementation, so maintenance is performed by changing generative pattern options. Maintenance patterns would be beneficial at lower development layers.

Design Pattern Rationale Graphs (DPRG) are used to understand the application of design patterns in existing code [Baniassad et al. 2003], acting much like the first use of fragments above. A DPRG presents a searchable text representation of the pattern that are linked to source code elements. This linking is simplified by allowing the tool to infer some of the links using pattern and source code knowledge. Once the pattern implementation is better understood, a maintainer can more effectively evolve the code. Again, CO_2P_3S is targeted at program development and not program understanding, so its patterns are displayed in the CO_2P_3S GUI. However, any patterns implemented manually in user code will not be documented. DPRGs could be useful for identifying and maintaining these patterns.

7.2.2 *Parallel and Distributed Tools.* In the parallel programming domain, there have been several design-pattern-based tool efforts (called *template-based* tools in early work). Systems of note include Enterprise [Schaeffer et al. 1993], DPnDP [Siu et al. 1996], Parallel Architectural Skeletons (PAS) [Goswami et al. 2002; 2001], and MAP_3S [Mehta et al. 2006]. Each of these provides abstractions to simplify the process of creating parallel applications based on a set of support parallel design patterns. Each of these systems target distributed-memory execution environments.

Enterprise is a distributed-memory tool based on graphically annotating sequential C code. The annotations indicate which procedures should be executed on remote processors and the communication structure between these processors. A precompiler transforms the input C code into parallel code based on the annotations, by transforming procedure calls to remote message sends to other processes. The set of available communication structures is based on a set of supported parallel design patterns, like master-worker (where the set of workers can be heterogeneous or homogeneous), divide-and-conquer, and the pipeline. For a pattern to be applied to user code, the code must contain a sequence of procedure calls that match the pattern selected for those procedures since no structural code is generated. For instance, in an Enterprise pipeline, the user code must contain a sequence of procedure calls that form the pipeline structure. Enterprise patterns do contain a small number of options for customization, mainly for optimization. These options are interface-neutral and can be changed at any time. Structural changes to an Enterprise pattern are generally limited to changing the set of procedures that are affected by the annotation. An example of this is adding a new procedure to a pipeline computation. This change may be interface-affecting because of calling conventions in Enterprise procedure calls; simple pointer data (arrays) can be passed between remote procedures, but they must be followed by an explicit size

argument to ensure data is properly marshaled. If this is not already part of the procedure, it must be added.

DPnDP (built on PVM [Geist et al. 1994]) and PAS (built with MPI [Snir et al. 1996]) provide high-level communication primitives to application developers. The selected pattern dictates the set of primitives available (some primitives are available only for specific patterns) and their behaviour. The supplied primitives automatically handle some design-pattern-specific communication problems (such as control messages between masters and workers), but all application-specific communication must be implemented by the user. One concern with maintaining DPnDP and PAS programs is that changes to the pattern options may change the semantics of message-passing primitives contained inside user code. As a result, changes to option values may require corresponding changes to user code to reflect the new behaviour.

In contrast to Enterprise, DPnDP, and PAS, the Patterns Layer of CO_2P_3S generates a complete pattern implementation and exports hook methods for user code. This features ensures that the pattern structure is correct. Further, it simplifies programming over DPnDP and PAS by not requiring users to write any parallel code; the hook methods are sequential code that contain no message passing or synchronization constructs. In terms of maintenance, of all of the systems mentioned in this section, Enterprise programs have the advantage. At the worst, changes to an Enterprise pattern option will only require a recompilation (to regenerate the code for the transformed procedure calls) since the program must already follow the design pattern structure. DPnDP may require user code be changed to account for new communication primitive semantics. A change to a CO_2P_3S generative pattern option may result in a new generated framework with new hook methods or new signatures for existing hook methods, which may require application changes.

RMA is a tool for developing pattern-based Java 2 Enterprise Edition (J2EE) web applications [Chen 2004; Chen and MacDonald 2005; 2006] using the design patterns and J2EE framework outlined in [Alur et al. 2003]. In terms of maintenance, the J2EE patterns in [Alur et al. 2003] tend to have fewer structural options that need to be supported by their generative counterparts, though those that do exist can be changed and code generated for the new pattern structure. Instead, the power of J2EE patterns is in their composition in the J2EE framework. RMA exploits the J2EE framework by generating not just the implementation code for individual patterns but also composing the patterns with related patterns in the framework. RMA is able to incorporate new patterns and remove unnecessary patterns during both development and subsequent maintenance, updating the remainder of the framework as needed. CO_2P_3S supports individual patterns since there is no complete application framework to build on; the composition of CO_2P_3S patterns is an application-level problem that cannot be automated.

MAP_3S (MPI Advanced Pattern-Based Parallel Programming System) uses CO_2P_3S-style program development in a distributed-memory environment, where programs are written in C using MPI. Hook methods are replaced with macros that are textually inserted into a generated C back-end implementing the selected design pattern. Since MAP_3S uses the same approach as CO_2P_3S, it shares the same maintenance characteristics described in this paper.

7.3 Aspect-oriented Approaches

A newer approach to implementing design patterns is to take advantage of *aspect-oriented programming* [Kiczales et al. 1997]. Aspect-oriented programming is a general method

for dealing with cross-cutting concerns by inserting code at well-defined points in the execution of a program. Different programming languages have tools extending the language with aspect-oriented mechanisms. For Java, one such extension is AspectJ [Kiczales et al. 2001]. AspectJ is based on two main ideas: *join points* and *advice*. A join point is a well-defined event in the execution of a program. For example, a join point may be a method call, a method return, or a field access. A *pointcut* assembles a set of join points. Advice is code that is attached to a pointcut and runs when the event occurs. The pointcuts and advice are encapsulated into an entity called an *aspect*. The aspect and application code are combined in a compile-time process called *weaving*.

Hannemann and Kiczales experimented with producing reusable aspects that implement design patterns using AspectJ, from which the programmer must derive a subaspect for its use in an application [Hannemann and Kiczales 2002]. The goals of these aspect-oriented versions were to increase modularity properties, separating the pattern implementation from the rest of the functionality in the class. However, each aspect represents a specific member of the pattern family. One of the goals was to be able to add or remove new patterns without affecting the rest of the classes by implementing all pattern responsibilities in the aspect and not in the class, which allows a pattern to be changed by replacing the advice. That is, the pattern implementation is isolated in a single entity, the aspect, and not scattered throughout the application code. However, AspectJ cannot make interface-affecting changes; such changes would span both the pattern subaspect and user code, and would need to be done manually. More specifically, AspectJ has the capability of adding new methods though *intertype declarations*, but cannot alter or remove existing methods in a class. Such changes are supported in CO_2P_3S. CO_2P_3S also provides some separation between user code and pattern code through the generated frameworks.

Aspect-oriented programming has also been applied to parallel programming, in an attempt to separate parallelism concerns from application code. One example tried to use AspectJ to introduce parallelism into sequential scientific Java code that consisted of independent, embarrassingly-parallel loops [Harbulot and Gurd 2004]. The aspects needed to split the loops in the programs into individual tasks and assign them to processors. However, the scientific code was not amenable to aspect-oriented programming since it was not written in an object-oriented fashion. In particular, the applications had to be refactored so the *iteration space* (the set of indices enumerated by a given set of loops) was available as method parameters. This is necessary to allow the aspect to intercept calls to the scientific computation, and access and manipulate the iteration space to split the loops. This refactoring step was necessary because pointcuts in AspectJ cannot be defined for individual statements within a method. The amount of refactoring needed depended on the individual application, but in some cases required extensive changes to the original application. The need for refactoring a program to make it amenable to aspect-oriented programming is not specific to scientific computing; it was also noted in an earlier exploratory study using AspectJ [Murphy et al. 2001]. CO_2P_3S programs may also require some refactoring if written in an imperative style. Specifically, CO_2P_3S frameworks work on individual data elements in a problem. If the sequential code does not explicitly represent operations on these elements as methods, then it will be difficult to reuse the existing code.

Chalabine and Kessler combine aspect weaving and invasive composition to parallelize sequential application code [Chalabine and Kessler 2005; 2006; 2007]. The key difference between aspect-oriented programming and invasive composition is that aspect-oriented programming is limited to inserting advice at fixed join points where invasive composi-

tion can rewrite the existing code. As a result, invasive programming can address concerns within a method and not just at method boundaries, and can replace application code with parallel code. These capabilities could be used to parallelize the scientific code described above without requiring the code be refactored first, by rewriting the loops in place to partition the iterations and distribute them across processors. In contrast, CO_2P_3S focuses on generating a parallel structure rather than altering a sequential version to include parallel concerns.

7.4 Other Approaches

LayOM implements a layered object model, where an object can be wrapped in a series of layers that augment its behaviour [Bosch 1998]. The layers intercept all incoming and outgoing method calls to alter or add functionality to the underlying object. Several layers have been defined that implement the core functionality of a set of design patterns. For example, for the Adapter pattern, a layer can be used to map method names from the adapter to the adaptee. For the Composite pattern, a layer can be added to include child management methods and to "multicast" methods in the composite class to all of its children. The choice of safety versus transparency is made by attaching the child management layer to the appropriate class. Adding and removing layers is a simple task. However, it is not clear if it is possible to easily customize a layer to implement design pattern variations during development or maintenance. It may be necessary to either use a different layer that implements the desired pattern variant (again incurring the difficulties noted in Section 3) or implement the pattern manually.

The ELIDE project allows objects to be annotated in a manner similar to **LayOM**, allowing user-defined modifiers to be added to classes, methods, and instance variables [Bryant et al. 2002]. These new modifiers can be used to generate additional code that is inserted into application classes. These modifiers can include parameter values that can be used to alter the generated code. Further, the code transformations for modifiers are implemented using Java classes provided by the user, and the output is standard Java code. This combination of features provides a more flexible and extensible system than **LayOM**, and may provide a mechanism to easily alter a pattern implementation during maintenance. This system was used to implement the Visitor and Flyweight patterns.

Metaprogramming provides another approach to implement design patterns in an application [von Dincklage 2003]. A metaprogram is a program that creates or manipulates another program. In this case, the programmer annotates classes with calls to metaprograms that generate the implementation of the selected design pattern. The metaprograms have access to information about the application classes and can accept parameters, making it more flexible than **LayOM** and similar to ELIDE.

One of the benefits to these approaches is that they make the use of patterns more explicit, helping with the design erosion problem [van Gurp and Bosch 2002]. The CO_2P_3S GUI makes the use of CO_2P_3S-supported patterns explicit. However, any additional patterns in user code must be documented.

7.5 Observations

An interesting observation in some research is that some patterns do not benefit from tool support. In [von Dincklage 2003], von Dincklage notes that metaprogramming does not help with the implementation of the Builder, Command, Factory Method, and Iterator patterns from [Gamma et al. 1994]. The argument is that a metaprogram cannot generate

a significant amount of the implementation. Hannemann and Kiczales try to better categorize which roles in a pattern benefit from their aspect-oriented approach [Hannemann and Kiczales 2002]. Roles are classified as either *defining roles* or *superimposed roles*. A defining role is a class that has no use outside of the pattern implementation. For example, in the Façade pattern, the façade generally redirects methods to objects in a subsystem and provides no additional functionality in the application [Gamma et al. 1994]. A superimposed role is a pattern responsibility that is given to a class that also has application responsibilities. For example, in the Observer pattern [Gamma et al. 1994], subjects and observers contribute application functionality as well as implementing the pattern. Patterns that consist of defining roles did not benefit from the aspect-oriented approach where those consisting of superimposed roles did. The benefits for patterns with both types of roles were pattern-dependent. Based on these results, it seems likely that there will be design patterns for which CO_2P_3S cannot offer much tool support, though we have not investigated this problem.

8. CONCLUSIONS

This paper showed that a design-pattern-based programming system based on structural code generation can reduce the difficulty and cost of revisiting and, most importantly, changing high-level design decisions. Too often, these decisions must be made before complete information is know about the application and its needs. Unfortunately, the results of these decisions become embedded into the application code, in both obvious and subtle ways. Locating and changing these decisions requires considerable development effort, and requires further testing and verification to ensure that the changes were done properly. With a design-pattern-based system capable of generating specialized code for different pattern structures, some of these decisions are embedded in code that is tool-generated. It is possible to regenerate this code for a different design pattern variant with minimal impact on application code when the changes are interface-neutral. For interface-affecting changes, the ability to generate code is even more important as the scope of changes to the structural code can be extensive.

To demonstrate this capability, we examined a parallel reaction-diffusion texture generation application written with the CO_2P_3S pattern-based parallel programming system, which implements the Parallel Design Patterns (PDP) process. Using this example, we showed that it is possible to automate changes to several important, interface-affecting design decisions that affect the communication structure of the parallel mesh computation. Normally, these changes would require considerable development and testing effort, but with appropriate design pattern support it is possible to quickly specify the new design and regenerate the structural framework code so that it uses the new interface to the user code. As well, we showed that the selection of values for interface-neutral options can be deferred as late as possible. In our case, the selection between shared-memory and distributed-memory architectures, normally a fundamental design decision in parallel programming, can be deferred until late in the development cycle and can be changed at any time with little impact on application code. This is possible even though both architectures have their own characteristics. This feature does not inhibit the ability of a system to generate code that is optimized for a particular architecture. Providing the ability to automate design changes will improve the flexibility of software and simplify maintenance, not just in parallel programming but in any application domain that benefits from pattern-based development tools.

In the future, we hope to find more ways to exploit design patterns in programming tools. We can see some progress in RoadMapAssembler [Chen 2004; Chen and MacDonald 2005; 2006], which uses pattern information to generate not just standalone pattern implementations but also associations between these patterns as dictated by the Java 2 Enterprise Edition (J2EE) framework in [Alur et al. 2003]. In addition, these relationships must be used to generate *deployment descriptors*, XML-based files that describe the application structure to the J2EE runtime system. Pattern information may also help improve support tools. Visualization systems may be able to exploit pattern information to provide more intuitive views of the software and map performance data back to the source code. Compilers may be able to better target their optimizations to expensive parts of the application. We believe that better use of pattern information in programming systems will only improve their abstractions and reduce programmer effort.

ACKNOWLEDGMENTS

We would like to thank the other members of the CO_2P_3S team from the University of Alberta, John Anvik [Anvik 2002], Steven Bromling [Bromling 2001], Patrick Earl [Earl 2004], and Zhuang Guo [Guo 2003] for their hard work over the lifetime of the project. The project has benefited immensely from all of their efforts.

We would also like to thank the reviewers for their excellent feedback. The paper has greatly benefited from their comments.

REFERENCES

ALUR, D., CRUPI, J., AND MALKS, D. 2003. *Core J2EE Patterns: Best Practices and Design Strategies*, Second ed. Sun Microsystems Press.

AMZA, C., COX, A., DWARKADAS, S., KELEHER, P., LU, H., RAJAMONY, R., YU, W., AND ZWAENEPOEL, W. 1996. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer 29,* 2, 18–28.

ANVIK, J. 2002. Evaluating generative parallel design patterns. M.S. thesis, Department of Computing Science, University of Alberta.

ANVIK, J., MACDONALD, S., SZAFRON, D., SCHAEFFER, J., BROMLING, S., AND TAN, K. 2002. Generating parallel programs from the wavefront design pattern. In *Proceedings of 7th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'02) (on CD–ROM)*. 8 pages.

Apache Jakarta Project 2005. *Velocity*. Apache Jakarta Project. http://jakarta.apache.org/velocity.

BANIASSAD, E., MURPHY, G., AND SCHWANNINGER, C. 2003. Design pattern rationale graphs: Linking design to source. In *Proceedings of the 25th International Conference on Software Engineering*. 352–362.

BOSCH, J. 1998. Design patterns as language constructs. *Journal of Object–Oriented Programming 11,* 2, 18–32.

BROMLING, S. 2001. Meta–programming with parallel design patterns. M.S. thesis, Department of Computing Science, University of Alberta.

BROMLING, S., MACDONALD, S., ANVIK, J., SCHAEFFER, J., SZAFRON, D., AND TAN, K. 2002. Pattern–based parallel programming. In *Proceedings of the 2002 International Conference on Parallel Processing (ICPP2002)*. 257–265.

BRYANT, A., CATTON, A., DE VOLDER, K., AND MURPHY, G. 2002. Explicit programming. In *Proceedings of First International Conference on Aspect-Oriented Software Development*. 10–18.

BUDINSKY, F., FINNIE, M., VLISSIDES, J., AND YU, P. 1996. Automatic code generation from design patterns. *IBM Systems Journal 35,* 2, 151–171.

CHALABINE, M. AND KESSLER, C. 2005. Parallelisation of sequential programs by invasive composition and aspect weaving. In *Proceedings of the 6th International Workshop on Advanced Parallel Processing Technologies*. Lecture Notes in Computer Science, vol. 3756. Springer-Verlag, 131–140.

CHALABINE, M. AND KESSLER, C. 2006. Crosscutting concerns in parallelization by invasive software composition and aspect weaving. In *Proceeding of the 39th Annual Hawaii International Conference on System Sciences*.

CHALABINE, M. AND KESSLER, C. 2007. A formal framework for automated round-trip software engineering in static aspect weaving and transformations. In *Proceedings of the 29th International Conference on Software Engineering*. 137–146.

CHEN, J. 2004. RMA: A pattern–based J2EE development tool. M.S. thesis, School of Computer Science, University of Waterloo.

CHEN, J. AND MACDONALD, S. 2005. RoadMapAssembler: A new pattern-based J2EE development tool. In *Proceedings of CASCON 2005*. 113–127.

CHEN, J. AND MACDONALD, S. 2006. Exploiting roles and responsibilities to generate code in a distributed design-pattern-based programming system. In *Proceedings of the 2006 International Conference on Parallel and Distributed Processing Techniques and Applications*. 17–23.

DIETZ, S., CHAMBERLAIN, B., AND SNYDER, L. 2004. Abstractions for dynamic data distribution. In *Proceedings of the 9th International Workshop on High–Level Parallel Programming Models and Supportive Environments (HIPS 2004)*. 42–51.

EARL, P. 2004. Types and code generation for use in generative design patterns. M.S. thesis, Department of Computing Science, University of Alberta.

FLORIJN, G., MEIJERS, M., AND VAN WINSEN, P. 1997. Tool support for object–oriented patterns. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*. Lecture Notes in Computer Science, vol. 1241. Springer–Verlag, 472–495.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1994. *Design Patterns: Elements of Reusable Object–Oriented Software*. Addison–Wesley.

GEIST, A., BEGUELIN, A., DONGARRA, J., JIANG, W., MANCHEK, R., AND SUNDERAM, V. 1994. *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press.

GOSWAMI, D., SINGH, A., AND PREISS, B. 2001. Building parallel applications using design patterns. In *Advances in Software Engineering: Topics in Comprehension, Evolution and Evaluation*. Springer–Verlag, Chapter 12, 243–265.

GOSWAMI, D., SINGH, A., AND PREISS, B. 2002. From design patterns to parallel architectural skeletons. *Journal of Parallel and Distributed Computing 62,* 4, 669–695.

GUO, Z. 2003. Developing network server applications using generative design patterns. M.S. thesis, Department of Computing Science, University of Alberta.

HAMMOUDA, I. AND HARSU, M. 2004. Documenting maintenance tasks using maintenance patterns. In *Proceeding of the 8th European Conference on Software Maintenance and Reengineering*. 37–47.

HAMMOUDA, I. AND KOSKIMIES, K. 2002. A pattern-based J2EE application development environment. *Journal of Nordic Computing 9,* 3, 248–260.

HAMMOUDA, I., KOSKINEN, J., PUSSINEN, M., KATARA, M., AND MIKKONEN, T. 2004. Adaptable concern-based framework specialization in UML. In *Proceedings of the 19th International Conference on Automated Software Engineering*. 78–87.

HANNEMANN, J. AND KICZALES, G. 2002. Design pattern implementation in Java and AspectJ. In *Proceedings of 17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 161–173.

HARBULOT, B. AND GURD, J. 2004. Using AspectJ to separate concerns in parallel scientific Java code. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development*. 122–131.

HighProductivity.org 2005. *High Productivity Computer Systems*. HighProductivity.org. http://www.-highproductivity.org.

JANSEN, A. AND BOSCH, J. 2004. Evaluation of tool support for architectural evolution. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering*. 375–378.

JANSEN, A. AND BOSCH, J. 2005. Software architecture as a set of architectural design decisions. In *Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture*. 109–120.

JOHNSON, R. AND FOOTE, B. 1988. Designing reusable classes. *Journal of Object–Oriented Programming 1,* 2, 22–35.

KICZALES, G., HILSDALE, E., HUGUNIN, J., KERSTEN, M., PALM, J., AND GRISWOLD, W. 2001. An overview of AspectJ. In *Proceedings of the Fifteenth European Conference on Object–Oriented Programming (ECOOP'01)*. Lecture Notes in Computer Science, vol. 2072. Springer–Verlag, 327–353.

KICZALES, G., LAMPING, J., MENDHEKAR, A., MAEDA, C., LOPES, C. V., LOINGTIER, J.-M., AND IRWIN, J. 1997. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object–Oriented Programming (ECOOP97)*. Lecture Notes in Computer Science, vol. 1241. Springer-Verlag, 220–242.

KLUSENER, S., LÄMMEL, R., AND VERHOEF, C. 2005. Architectural modifications to deployed software. *Science of Computer Programming 54*, 143–211.

LU, P. 2000. Implementing scoped behaviour for flexible distributed data sharing. *IEEE Concurrency 8,* 3 (July–September), 63–73.

MACDONALD, S. 2002. From patterns to frameworks to parallel programs. Ph.D. thesis, Department of Computing Science, University of Alberta.

MACDONALD, S., ANVIK, J., BROMLING, S., SCHAEFFER, J., SZAFRON, D., AND TAN, K. 2002. From patterns to frameworks to parallel programs. *Parallel Computing 28,* 12, 1663–1683.

MACDONALD, S., SZAFRON, D., AND SCHAEFFER, J. 1999. Object–oriented pattern–based parallel programming with automatically generated frameworks. In *Proceedings of the 5th USENIX Conference on Object–Oriented Technology and Systems (COOTS'99)*. 29–43.

MACDONALD, S., SZAFRON, D., SCHAEFFER, J., ANVIK, J., BROMLING, S., AND TAN, K. 2002. Generative design patterns. In *Proceedings of 17th IEEE International Conference on Automated Software Engineering (ASE2002)*. 23–34.

MACDONALD, S., SZAFRON, D., SCHAEFFER, J., AND BROMLING, S. 2000. Generating parallel program frameworks from parallel design patterns. In *Proceedings of the 6th International Euro–Par Conference*. Lecture Notes in Computer Science, vol. 1900. Springer–Verlag, 95–104.

MATTSON, T., SANDERS, B., AND MASSINGILL, B. 2004. *Patterns for Parallel Programming*. Addison–Welsey.

MCNAUGHTON, M., REDFORD, J., SCHAEFFER, J., AND SZAFRON, D. 2003. Pattern-based AI scripting using ScriptEase. In *Proceeding of the Sixteenth Canadian Conference on Artificial Intelligence*. 35–49.

MEHTA, P., AMARAL, J., AND SZAFRON, D. 2006. Is MPI suitable for a generative design-pattern system? *Parallel Computing 32,* 7-8, 616–626.

MURPHY, G., LAI, A., WALKER, R., AND ROBILLARD, M. 2001. Separating features in source code: An exploratory study. In *Proceedings of the 23rd International Conference on Software Engineering*. 275–284.

Object Technology International, Inc. 2006. *Eclipse Platform Technical Overview*. Object Technology International, Inc. Available at: http://www.eclipse.org/articles/Whitepaper-Platform-3.1/eclipse-platform-whitepaper.html.

POLLACK, M. 2000. Code generation using JavaDoc. http://www.javaworld.com/javaworld/jw-08-2000/jw-0818-javadoc_p.html.

SCHACH, S. 2007. *Object-Oriented and Classical Software Engineering*, Seventh ed. McGraw-Hill.

SCHAEFFER, J., SZAFRON, D., LOBE, G., AND PARSONS, I. 1993. The enterprise model for developing distributed applications. *IEEE Parallel and Distributed Technology 1,* 3, 85–96.

SCHMIDT, D., STAL, M., ROHNERT, H., AND BUSCHMANN, F. 2000. *Pattern–Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. Vol. 2. Wiley & Sons.

SIIKARLA, M. AND SYSTÄ, T. 2006. Transformational pattern system - some assembly required. In *Proceedings of the 5th International Workshop on Graph Transformation and Visual Modeling Techniques*. 57–68.

SINGH, A., SCHAEFFER, J., AND SZAFRON, D. 1998. Experience with parallel programming using code templates. *Concurrency: Practice and Experience 10,* 2, 91–120.

SIU, S., SIMONE, M. D., GOSWAMI, D., AND SINGH, A. 1996. Design patterns for parallel programming. In *Proceedings of the 1996 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'96)*. 230–240.

SNIR, M., OTTO, S., HESS-LEDERMAN, S., WALKER, D., AND DONGARRA, J. 1996. *MPI: The Complete Reference*. MIT Press.

Sun Microsystems Inc. 2001. *Jini Architectural Overview*. Sun Microsystems Inc. http://wwws.sun.com/-software/jini/whitepapers/architecture.pdf.

TAN, K. 2003. Pattern–based parallel programming in a distributed memory environment. M.S. thesis, Department of Computing Science, University of Alberta.

TAN, K., SZAFRON, D., SCHAEFFER, J., ANVIK, J., AND MACDONALD, S. 2003. Using generative design patterns to generate parallel code for a distributed memory environment. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2003)*. 203–215.

Team in a Box Ltd. 2002. *Eclipse Metrics Plug-in*. Team in a Box Ltd. http://www.teaminabox.co.uk/downloads/-metrics/index.html.

VAN GURP, J. AND BOSCH, J. 2002. Design erosion: Problems and causes. *Journal of Systems and Software 61,* 2, 105–119.

VAN WINSEN, P. 1996. (re)engineering with object-oriented design patterns. M.S. thesis, Department of Information and Computing Sciences, Utrecht University.

VILJAMAA, A. AND VILJAMAA, J. 2002. Creating framework specialization instructions for tool environments. In *Proceedings of the Nordic Workshop on Software Development Tools and Techniques*.

VLISSIDES, J. 1998. *Pattern Hatching: Design Patterns Applied*. Addison–Wesley.

VON DINCKLAGE, D. 2003. Making patterns explicit with metaprogramming. In *Proceedings of the Second International Conference on Generative Programming and Component Engineering*. Lecture Notes in Computer Science, vol. 2830. Springer-Verlag, 287–306.

WITKIN, A. AND KASS, M. 1991. Reaction–diffusion textures. *Computer Graphics (SIGGRAPH '91 Proceedings) 25,* 4, 299–308.

Received February 2005; revised December 2006; accepted June 2007