IMPLEMENTATION OF A PARALLEL PROGRAM, PROGRAM GENERATOR

By

JEAN-DAVID G. OLADÉLE

A THESIS PRESENTED TO THE GRADUATE SCHOOL OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF ENGINEERING

UNIVERSITY OF FLORIDA

2002
Dedicated to Renee Williams.
ACKNOWLEDGMENTS

I thank my Lord Jesus Christ for his mercy and strengthening me in all that I do. I would like to express my sincere gratitude to my advisor, Dr. Beverly Sanders, for serving as the chairperson of my committee. Her suggestions and support went a long way in making this thesis a reality. I would also like to thank Dr. Hammer and Dr. Wilson for their commitments.

I thank Alejandro Pauly for his examples that provided much needed insight during the early stages of my work. In addition I am grateful to all my friends for their support and encouragement. Finally, I would like to express my love and gratitude to my family for their support and dedication.
# TABLE OF CONTENTS

ACKNOWLEDGMENTS ................................................................................................. iv

LIST OF TABLES ........................................................................................................... vii

LIST OF FIGURES ......................................................................................................... viii

ABSTRACT ......................................................................................................................... x

CHAPTER

1 INTRODUCTION ........................................................................................................... 1

2 BACKGROUND ............................................................................................................. 4

  Program Generators ..................................................................................................... 4
    The OO-Driven Style ................................................................................................. 5
    Code-Driven Style ................................................................................................. 6
    Table-Driven Style ................................................................................................. 6
  Patterns, Pattern Languages and Parallel Patterns .................................................... 7
  Extensible Markup Language and Data Type Definitions ........................................ 10
  Document Object Model ........................................................................................... 10

3 THE PPPG USER SPECIFICATIONS ....................................................................... 11

  The design.space Element ....................................................................................... 12
  The task.group Element ........................................................................................... 13
  The grp.struct Element ........................................................................................... 13
  The grp.sub.struct Element .................................................................................... 13
  The task.decomposition Element .......................................................................... 14
  The task.relationships Element ............................................................................. 14
  The dependency.type Element ............................................................................... 14

4 USING THE PPPG TOOL .......................................................................................... 22

  The PPPG User Interface .......................................................................................... 22
    General Information Pane .................................................................................... 22
    Datatypes Pane ....................................................................................................... 23
    Concurrency Organization Pane ........................................................................... 24
    Dependencies and Partitioning Pane .................................................................... 25
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3-1</td>
<td>design.space Element summary</td>
<td>16</td>
</tr>
<tr>
<td>3-2</td>
<td>task.group Element summary</td>
<td>17</td>
</tr>
<tr>
<td>3-3</td>
<td>grp.struct Element summary</td>
<td>18</td>
</tr>
<tr>
<td>3-4</td>
<td>grp.sub.struct Element summary</td>
<td>18</td>
</tr>
<tr>
<td>3-5</td>
<td>task.decomposition Element summary</td>
<td>19</td>
</tr>
<tr>
<td>3-6</td>
<td>task.relationships Element summary</td>
<td>19</td>
</tr>
<tr>
<td>3-7</td>
<td>dependency.type Element summary</td>
<td>20</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-1 Algorithm structure decision tree</td>
<td>8</td>
</tr>
<tr>
<td>3-1 The design.space Element</td>
<td>16</td>
</tr>
<tr>
<td>3-2 The task.group Element</td>
<td>17</td>
</tr>
<tr>
<td>3-3 The grp.struct Element</td>
<td>17</td>
</tr>
<tr>
<td>3-4 The grp.sub.struct Element</td>
<td>18</td>
</tr>
<tr>
<td>3-5 The task.decomposition Element</td>
<td>18</td>
</tr>
<tr>
<td>3-6 The task.relationships Element</td>
<td>19</td>
</tr>
<tr>
<td>3-7 The dependency.type Element</td>
<td>19</td>
</tr>
<tr>
<td>3-8 The design.space element code snippet</td>
<td>20</td>
</tr>
<tr>
<td>3-9 The task.group element code snippet</td>
<td>20</td>
</tr>
<tr>
<td>3-10 The grp.struct element code snippet</td>
<td>20</td>
</tr>
<tr>
<td>3-11 Complete specification document</td>
<td>21</td>
</tr>
<tr>
<td>4-1 General information screen</td>
<td>23</td>
</tr>
<tr>
<td>4-2 Datatypes window</td>
<td>24</td>
</tr>
<tr>
<td>4-3 Concurrency organization pane</td>
<td>25</td>
</tr>
<tr>
<td>4-4 Dependencies and partitioning pane</td>
<td>26</td>
</tr>
<tr>
<td>4-5 The XML Viewer Pane</td>
<td>27</td>
</tr>
<tr>
<td>5-1 Specification file for mergesort procedure</td>
<td>29</td>
</tr>
<tr>
<td>5-2 Decision tree</td>
<td>31</td>
</tr>
<tr>
<td>5-3 Boilerplate code generated to mergeSort.java file</td>
<td>32</td>
</tr>
</tbody>
</table>
5-4 Remainder of code generated for mergeSort.java class ..............................................34
5-4 Structure of PPPG ...........................................................................................................36
5-5 Parse and package module ..........................................................................................37
5-6 SyntaxAnalyzer UML diagram .....................................................................................38
5-7 DataCapsule UML diagram .........................................................................................39
5-8 CodeGenerator UML diagram .....................................................................................40
6-1 Concurrent MergeSort algorithm ....................................................................................42
6-2 Implementation of the split method ............................................................................43
6-3 MathFactory code snippet ..........................................................................................44
6-4 Inner Class Stage code snippet ..................................................................................45
6-5 Game of life ..................................................................................................................47
Efficient parallel programs and algorithms are difficult to write. Parallel programs present the programmer not only with the questions of what needs to be done, and when an operation takes place, but also where the operation needs to occur. A large number of parallel programs are dynamic in nature and do not provide a regular structure for parallelization. The programmer must provide complex algorithms for scheduling, load balancing, and rebalancing in order for these programs to perform efficiently in parallel.

This thesis presents the design and implementation of the PARALLEL PROGRAM, PROGRAM GENERATOR (PPPG). PPPG reduces the complexity of creating parallel programs. PPPG generates efficient and portable parallel patterns for different classes of the parallel program domain. The patterns generated are dependent on Extensible Markup Language (XML) input specifications. The input specifications may be supplied as a raw XML file or generated from a series of responses by the user.
The output patterns are provided in the Java language to allow for portable parallel programs. The Java Virtual Machine (JVM) is responsible for processor scheduling.
CHAPTER 1
INTRODUCTION

Parallel programming involves breaking an algorithm or data into parts. These parts are then distributed as tasks for simultaneous execution over multiple processors or concurrent execution over a single processor. The motivation for parallel programming is that a program being executed over multiple processors might execute faster than it would during serial execution on a single processor. Parallel programming involves design steps to assign threads, tasks, or processes to physical processor(s) for execution [6]. Creating and writing parallel programs is often very difficult and time consuming. The programmer must deal with issues of correctness and efficiency, which become more complex and magnified in a parallel environment. Dependencies among concurrent threads of execution must be carefully managed to ensure a consistent result.

The aforementioned difficulties necessitate a tool that can generate a shell program or template that makes writing parallel programs easier for the developer. This tool will handle all the concurrency and performance issues. The generated shell program is essentially ready for use. The developer need only modify the template by adding application-specific functionality, and the program is ready for use.

This thesis describes the design and implementation of a program generator that provides a template for the creation of efficient and portable parallel programs. The program generator is referred to as the Parallel Program, Program Generator (PPPG).

A program generator is a tool that automates the creation of programs from precise descriptions of syntax and desired behavior [1]. Program generators provide an
effective way to automatically create variant programs within the same family. They allow for consistency of commonalities within a particular application domain. When there is sufficient information about how a program will be used, program generators construct optimized and robust code.

The PPPG generates shell programs for different classes of the parallel program domain. The generated shell/skeleton programs are based on the results of ongoing research at the University of Florida. This research focuses on developing pattern language for parallel application programming. Patterns are defined as representations of reusable object design. Patterns and Pattern languages are ways to describe best practices and good designs, and to capture experience in a way that allows others to reuse this experience [4]. Parallel patterns can be used to design high quality software because the pattern language acts as a guide through application design and the development process [7].

The PPPG requires information from the user in order to determine the characteristics, behavior, and class of parallel program to be generated. This information is called the input specification. The input specification for PPPG is represented using the Extensible Markup Language (XML). The XML is a cross-platform, software and hardware-independent language for data description and information transmission. The XML is used to help accomplish the PPPG’s goal of portability. The information representing the parallel programming domain is called the Document Type Definition (DTD). The DTD lists all the element types and attributes and their constraints as used in the input specification. A valid XML input specification document must meet the conditions imposed by the DTD.
The skeleton program generated by PPPG is written in the Java programming language. The main advantage of having the code generated in Java is portability over different and diverse operating environments. An added benefit is that the user need not worry about scheduling issues; the Java Virtual Machine (JVM) will handle this.

Using XML and Java, a program generator that is efficient and portable can be created. This development tool takes much of the burden of designing and implementing parallel programs off of the user. The remainder of this thesis paper describes the implementation of such a tool. Chapter 2 gives background information on the technologies behind PPPG. Chapter 3 discusses the DTD for the domain and how it constrains the XML specification. Chapter 4 provides a high-level view of the PPPG tool from the user’s perspective and addresses the effective use of the tool. Chapter 5 deals with implementation details and issues, from a designer’s point of view. Chapter 6 discusses sample programs based on patterns generated using PPPG. Chapter 7 gives conclusions and suggestions for future additions and improvements.
CHAPTER 2
BACKGROUND

This chapter presents an overview of the core development technology used to implement PPPG. This includes a brief review of program generators, patterns and pattern languages, the extensible markup language (XML), and data type definitions (DTD). This chapter also discusses parallel patterns and the parallel pattern language developed at University of Florida (UF).

Program Generators

A program generator is a software tool that creates the source code for another computer program. The description of what the generated computer program should do is supplied to the program generator as a specification file. Data about the program to be generated is supplied to the program generator, which after analyzing and transforming the data generates the desired program. Some popular examples of program generators are parser generators such as yacc, GUI builders, and visual programming tools.

Using program generators can greatly increase productivity. One reason for this is that specifications are usually easier to create, edit, and verify correctness than the code that implements the specifications. This is often the case because implementation often requires specialized knowledge and techniques that are very time consuming. These techniques are usually not required to understand and create the specifications. When a program generator is being used the programmer need only be concerned with the specification; without the program generator, a significant amount of time would also have been spent on implementation.
Program generators also increase productivity because they can easily be used to generate variant programs within the same problem domain or program family. Program generators can generate other files, such as documentation about the software, user documentation, test scripts and more. They can also resolve the consistent-update problem. During software maintenance or update, fixing an error may often set off a chain of other errors within the software product. This occurs often in situations where the change is not updated consistently across the software product. When using a program generator, the specification need only be updated and the program can be regenerated. Doing this will ensure consistent update across the application. In cases where information on how the generated program will be used is available, program generators can be specifically customized to generate optimized code.

There are 3 main styles of programming used in program generators. These styles relate directly to the type of code generated by the program generator. The three approaches are

- **OO-driven**
- **Code-driven**
- **Table-driven**

**The OO-Driven Style**

The OO-Driven approach uses standard object oriented techniques such as inheritance, interfacing, and message passing, for organizing the structure of the generated program [2]. Related variables and methods are bundled to create specific objects. Data from specifications are inserted where needed and Classes are created for attributes where needed. This approach is based on stacking object-oriented layers of
abstraction. Each layer contains a number of object classes where each top layer refines the layer below it by adding new classes and new methods to existing classes [3].

This particular style is usually difficult to create and is less efficient than the code-driven style. The advantage of this style is that it allows for separation of concerns based on specific performance and design goals.

**Code-Driven Style**

This is the most natural and simplistic approach to program generation. In this method the code is simply generated where it is needed without any concerns about program structure but focuses on simplicity and efficiency of code generation [2]. In this method data from the specification is simply embedded where needed.

**Table-Driven Style**

Table-driven design is a software engineering methodology that attempts to simplify and generalize by separating program control variable and parameters from the program code and placing them in external tables. Design objectives include an emphasis on modularity and decoupling program control data from application logic. Applications are made more flexible by postponing the time when control values and rules are bound to the processes they direct [11].

This approach to program generation emphasizes the separation between data and the code that performs the generation. Data from the specifications are stored in the tables and other data structures. The code simply refers to the tables when the data is needed rather than embedding the data directly into the code [2].

PPPG uses a hybrid of the OO and Code-driven approaches for program generation. Combining these 2 approaches results in simple and efficient code generation with attributes represented as objects.
Patterns, Pattern Languages and Parallel Patterns

Patterns are reusable representations of good design, which assist programmers in the development of good quality software. A collection of patterns is referred to as a pattern language. Pattern languages can be used as a guide from problem specifications to complete design [7].

PPPG will generate source code based on parallel patterns from ongoing research at the University of Florida. The research focuses on developing a pattern language for parallel application programming. The pattern language is to be used by application programmers who are interested in designing programs that will execute in parallel. The pattern language is organized around the following design spaces:

- Finding concurrency
- Algorithm structure
- Supporting structures
- Implementation mechanisms

This thesis focuses on patterns from the algorithm structure design space. This space deals with methods for exploiting concurrency. The patterns within this space are organized by means of the decision tree shown in Figure 2-1.

Starting at the root of the tree the program generator makes decisions based on the input specification and generates one of the 9 terminal patterns. The terminal patterns are discussed below:

**Pipeline Processing.** This pattern is used in situations where the data requires a number of basic operations to be performed in sequence. The operations may be regarded as stages or stops within a pipeline, where all the stages have the potential of executing concurrently. At each pipeline stage an operation is performed and then the data moves on to the next sequential stage. A good example of pipeline processing is an assembly
line at an electronics manufacturer. An unfinished VCR or DVD player may go through different line stages before completion. At each stage the same operation is performed over and over again.

Asynchronous Composition. This pattern is applicable in situations where units may interact with each other in a random non-specific manner.

Embarrassingly Parallel. The embarrassingly parallel pattern is applicable in situations where various units can operate without any interaction. This pattern usually involves the concurrent execution of independent units. This pattern shows how to organize such units so that they may execute in an efficient manner.

Reduction. The reduction pattern is used in situations where dependencies between tasks can be removed based on whether the global data can be copied to individual tasks and
partial results stored in these tasks can be recombined to form a global solution. This pattern represents the class of problems in which the following issues can be separated.

- Allocating the tasks among the processors so the computational load is evenly distributed.
- Managing the dependencies between tasks so that if multiple tasks update the same data structure, these updates do not interfere with each other.

**Shared Memory.** The Shared Memory pattern is used in situations where dependencies between tasks cannot be removed. The two main challenges that this pattern helps the programmer deal with are, the partitioning of tasks amongst units of execution in such a manner that the computation load on the processor(s) is evenly distributed, and the management of any dependencies that may exist between tasks.

**Balanced Tree.** This pattern applies to problems in which a balanced tree can be built on the input elements. The pattern will allow the tree to be traversed in forward and backward directions while computing and storing information at each internal node [3].

**Divide and Conquer.** This pattern applies to problems where the input can be partitioned into individual tasks. Each of these tasks can be solved recursively and in parallel (concurrently) and the solutions of the different sub problems can be merged into the solution for the original problem.

**Geometric Decomposition.** This pattern is applicable when the initial problem can stored in a data structure which can be broken down into separate pieces. The concurrency is achieved based on parallel updates of the individual pieces. When any piece is updated data is required from all the other pieces.

**Reference Following.** This pattern is used for parallel applications in which an apparently sequential operation on a recursive data is reworked to make it possible to operate on all elements of the data structure concurrently [7].
Extensible Markup Language and Data Type Definitions

Extended Markup Language (XML) provides a way to define, manipulate, transform and transport data over multiple platforms. XML provides a way to define an arbitrary structure for documents. It has the advantage of keeping structure and presentation information separate. This in turn leads to more flexible and thus more valuable documents.

There are two types of correct XML documents, valid XML documents and well-formed XML documents. A well-formed XML document contains just the needed information. They do not spell out their own rules but simply follow the general XML rules. Valid XML documents on the other hand are self-describing and contain or provide a link to information about the documents structure and contents. These documents require a document type description (DTD). The DTD constrains the XML document for the application program that is to process the XML.

Document Object Model

The document object model (DOM) is a W3C specification described as follows:

The Document Object Model is a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents. The document can be further processed and the results of that processing can be incorporated back into the presented page[13].

The DOM provides standard interfaces to allow for programmatic access to the contents of an XML document. An XML document is represented as a tree data structure in which the nodes represent portions of the document. Using DOM, specific information can be retrieved from the XML document.
CHAPTER 3
THE PPPG USER SPECIFICATIONS

In order for PPPG to create useful and valuable programs for the user, there must be a means by which the user can clearly describe the characteristics and functionality of the program to be generated. The user must be able to easily and unambiguously communicate what kind of program is needed and the program generator must be able to understand the users’ description and generate the correct program. XML is used to create a specification language that adequately addresses the aforementioned goals. The rest of this chapter will focus on the XML that specifies the program to be generated and the DTD that describes and constrains it.

The primary goal of the specifications is to help the user describe his/her target program in terms of one of the terminal patterns shown in Figure 2-1. In other words, the specification language fits the user’s initial problem to the closest related terminal pattern. To adequately express what kind of program is to be generated the programmer must be able to answer the following questions about their problem:

- What kind of top level organization is desired for concurrency?
- What kind of structure does the top-level organization possess?
- Do any dependencies exist amongst tasks or task groups?

Let us now examine the individual elements and their corresponding attributes that make up the document type description for the PPPG specifications. This document type description will be referred to as the design space DTD.
The design space DTD contains the following elements in hierarchical order:

- design.space
- task.group
- grp.struct
- grp.sub.struct
- task.decomposition
- task.relationships
- dependency.type

The design.space Element

The design.space element serves as the root element for the design space DTD. Every other element is a descendant of this element. The design.space element has five attributes and one sub element. The attributes are listed below:

- **Classname.** This particular attribute represents the name of the Java class for which code will be generated. This is a required attribute and must be provided in the specification document. The required data type for this element is string.

- **Sourcepath.** This attribute represents the target directory in which the generated code should be stored. This attribute is optional. If it is not provided the generated code will be stored in the directory PPPG is being executed in. The required data type for this attribute is string.

- **Units.** This attribute indicates the number of units of execution that the user wants the generated program to use in the solution. This attribute is optional. If not specified a default value of three units is used. The required data type for this attribute is the primitive type int.

- **Initdatatype.** This attribute should contain the data type of the initial problem. This is an optional attribute and if not supplied the default data type Object is used. This attribute should be specified as a string.

- **Findatatype.** This attribute indicates the data type in which the user expects the solution to the problem to be stored. As in the case of the previous attribute, findatatype is also optional and a default data type of Object is assumed when the attribute is missing. This attribute should be specified as a string.

The design space element must contain an instance of the task.group element. A diagram of the design.space element is shown in Figure 3-1.
The task.group Element

The task.group element captures information about the top-level organization of the problem to be solved. It has two attributes and one sub-element. The attributes are described below:

- **Name.** This attribute should contain the name of the top-level organization for the problem. There are three acceptable values for this attribute. The values are Ordering, Tasks, and Data. This is a required attribute and must be provided in the specification document. The required data type for this element is string.

- **Datatype.** This attribute should contain the data type of the sub-problems that the initial problem will be broken down into. This attribute is optional. If the attribute is not supplied the data type Object is used. This attribute should be specified within specification document as a string.

The task group element requires a grp.struct sub-element.

The grp.struct Element

The grp.struct element helps to capture information about the underlying structure of a given top-level organization. It allows the user to provide information on how a problem with a given organization (orderings, tasks, or data) is composed. It consists of two attributes and a required sub element. The attributes are shown below:

- **Name.** This attribute contains the type of underlying structure for a specific top-level organization type. If the top-level organization is by Ordering the possible values are Regular or Irregular. For any other valid top-level organization the valid values are either Linear or Recursive. This attribute is required and must be specified as a string.

- **Numstages.** This attribute is only applicable when the problem is organized by ordering and has a regular structure. This attribute allows the user to specify the number of stages required. This is an optional attribute. The required data type for this attribute is the primitive type int.

The grp.sub.struct Element

The grp.sub.struct element captures information about tasks relationships and dependencies. It is only applicable in problems that are organized by tasks. It contains no
attributes but it must have either a task.decomposition or a task.relationships element as a sub element.

The task.decomposition Element

The task.decomposition element is applicable when we have a problem that is organized by tasks and has a recursive substructure. The possible values for this element are one-way and two-way. These values indicate the type of recursion the problem exhibits.

The task.relationships Element

The task.relationships element is applicable only when the top-level organization is by tasks. This element captures dependency information among linearly partitioned tasks. It contains a single required attribute and an optional sub-element. The attribute is described below:

- **Type.** This attribute indicates whether or not dependencies exist among the partitioned tasks. This attribute has two possible correct values. It can either be Dependent or Independent. This is a required attribute and must be provided in the specification document. The required data type for this element is string.

The dependency.type Element

The dependency.type element simply expresses what kind of dependencies are exhibited by the tasks. There are two valid options for this element. It can be either Separable or Inseparable.

The design space DTD elements that have been discussed help constrain and describe the XML specifications for the PPPG application. In this section the DTD elements will be transformed into the corresponding XML specification.

Consider the following example, a user wishes to write a parallel version of the game of life. The game should use no more than nine threads and the initial problem is
represented in a two dimensional integer array. The solution should be in a class with the name Life.java. Using the information given above, the XML specifications for the presented scenario will be created according to the design space DTD elements.

Starting with the design.space root element the user must identify the required attributes from the problem description: The class name is “Life”; the generated file should be stored in the user’s home directory, it should use nine units of execution and the problem is represented in a two dimensional array. Our initial specification document with the information above will look like this:

The design space DTD requires that a design.space element must also contain a task.group element. For the game of Life problem the top-level organization is by tasks. The user has also chosen to create a user-defined class named Task to contain the information on which part of the array each unit of execution will have access to. Adding this information to our specification file will result in Figure 3-9.

The task.group element requires that a grp.struct element be present. The user must focus on what attributes are needed for this element. From our specifications it can be deduced that the Tasks substructure is linear. In this particular scenario the numstages attribute is not applicable. Adding this information results in the specification file shown in Figure 3-10.

Now that information has been captured about the grp.struct we must also examine what information can be extracted from the problem description relating to the grp.sub.struct element. The grp.sub.struct element must contain either a task.relationships or task.dependency element type. The tasks in this problem are independent of one another and the task.relationships element is applicable in this
particular case. Adding this information results in the completed specification shown in Figure 3-11.

Using the design space DTD we have been able to transform the game of life problem description into an XML specification file that conveys all the necessary information needed to generate the source code for the problem solution.

The design space DTD has allowed us to fit the details of a problem to one of the decision tree terminal patterns. In doing so, we have been able to create accurate and concise specifications for the PPPG application.

Figure 3-1  The design.space Element

Table 3-1  design.space Element summary

<table>
<thead>
<tr>
<th>Element Name</th>
<th>design.space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attributes</td>
<td>classname, sourcepath, unit, initdatatype, findatatype</td>
</tr>
<tr>
<td>Uses element(s)</td>
<td>task.group</td>
</tr>
<tr>
<td>Used by</td>
<td>(None) root element</td>
</tr>
<tr>
<td>Content</td>
<td>a sequence of elements</td>
</tr>
<tr>
<td>Occurrence</td>
<td>required</td>
</tr>
</tbody>
</table>
Table 3-2 task.group Element summary

<table>
<thead>
<tr>
<th>Element Name</th>
<th>task.group</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attributes</td>
<td>name, datatype</td>
</tr>
<tr>
<td>Uses element(s)</td>
<td>grp.struct</td>
</tr>
<tr>
<td>Used by</td>
<td>design.space</td>
</tr>
<tr>
<td>Content</td>
<td>a sequence of elements</td>
</tr>
<tr>
<td>Occurrence</td>
<td>required</td>
</tr>
</tbody>
</table>

Figure 3-3 The grp.struct Element
Table 3-3  grp.struct Element summary

<table>
<thead>
<tr>
<th>Element Name</th>
<th>grp.struct</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attributes</td>
<td>name, numstages</td>
</tr>
<tr>
<td>Uses element(s)</td>
<td>grp.sub.struct</td>
</tr>
<tr>
<td>Used by</td>
<td>task.group</td>
</tr>
<tr>
<td>Content</td>
<td>a sequence of elements</td>
</tr>
<tr>
<td>Occurrence</td>
<td>required</td>
</tr>
</tbody>
</table>

Figure 3-4  The grp.sub.struct Element

Table 3-4  grp.sub.struct Element summary

<table>
<thead>
<tr>
<th>Element Name</th>
<th>grp.sub.struct</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attributes</td>
<td>None</td>
</tr>
<tr>
<td>Uses element(s)</td>
<td>task.decomposition task.relationships</td>
</tr>
<tr>
<td>Used by</td>
<td>grp.struct</td>
</tr>
<tr>
<td>Content</td>
<td>choice</td>
</tr>
<tr>
<td>Occurrence</td>
<td>required</td>
</tr>
</tbody>
</table>

Figure 3-5  The task.decomposition Element
Table 3-5  task.decomposition Element summary

<table>
<thead>
<tr>
<th>Element Name</th>
<th>task.decomposition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attributes</td>
<td>None</td>
</tr>
<tr>
<td>Uses element(s)</td>
<td>None</td>
</tr>
<tr>
<td>Used by</td>
<td>grp.sub.struct</td>
</tr>
<tr>
<td>Content</td>
<td>character data</td>
</tr>
<tr>
<td>Occurrence</td>
<td>optional</td>
</tr>
</tbody>
</table>

Figure 3-6  The task.relationships Element

Table 3-6  task.relationships Element summary

<table>
<thead>
<tr>
<th>Element Name</th>
<th>task.relationships</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attributes</td>
<td>type</td>
</tr>
<tr>
<td>Uses element(s)</td>
<td>dependency.type</td>
</tr>
<tr>
<td>Used by</td>
<td>grp.sub.struct</td>
</tr>
<tr>
<td>Content</td>
<td>a sequence of elements</td>
</tr>
<tr>
<td>Occurrence</td>
<td>required</td>
</tr>
</tbody>
</table>

Figure 3-7  The dependency.type Element
Table 3-7  dependency.type Element summary

<table>
<thead>
<tr>
<th>Element Name</th>
<th>dependency.type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attributes</td>
<td>None</td>
</tr>
<tr>
<td>Uses element(s)</td>
<td>None</td>
</tr>
<tr>
<td>Used by</td>
<td>task.relationships</td>
</tr>
<tr>
<td>Content</td>
<td>character data</td>
</tr>
<tr>
<td>Occurrence</td>
<td>optional</td>
</tr>
</tbody>
</table>

```
<?xml version = "1.0" encoding = "UTF-8"?>
<DOCTYPE design.space SYSTEM "http://www.cise.ufl.edu/~jolaedele/PPG/taskDecomposition.dtd">
<design.space classname = "Life" sourcepath = "G:\My Projects\Thesis\Work\GuiOut" intdatatype = "int[ ]" units = "9">
  <design.space>
    <design.space classname = "Life" sourcepath = "G:\My Projects\Thesis\Work\GuiOut" intdatatype = "int[ ]" units = "9">
      <task.group name = "Tasks" datatpe = "Task">
        <grp.struct name = "Linear">
        </grp.struct>
      </task.group>
    </design.space>
  </design.space>
</design.space>
```

Figure 3-8  The design.space element code snippet

```
<?xml version = "1.0" encoding = "UTF-8"?>
<DOCTYPE design.space SYSTEM "http://www.cise.ufl.edu/~jolaedele/PPG/taskDecomposition.dtd">
<design.space classname = "Life" sourcepath = "G:\My Projects\Thesis\Work\GuiOut" intdatatype = "int[ ]" units = "9">
  <task.group name = "Tasks" datatpe = "Task">
  </task.group>
</design.space>
```

Figure 3-9  The task.group element code snippet

```
<?xml version = "1.0" encoding = "UTF-8"?>
<DOCTYPE design.space SYSTEM "http://www.cise.ufl.edu/~jolaedele/PPG/taskDecomposition.dtd">
<design.space classname = "Life" sourcepath = "G:\My Projects\Thesis\Work\GuiOut" intdatatype = "int[ ]" units = "9">
  <task.group name = "Tasks" datatpe = "Task">
    <grp.struct name = "Linear">
    </grp.struct>
  </task.group>
</design.space>
```

Figure 3-10  The grp.struct element code snippet
Figure 3-11  Complete specification document
CHAPTER 4
USING THE PPPG TOOL

The PPPG tool provides two options for the user to generate source code for concurrent/parallel programs. Users may choose to generate their source code by supplying an XML specification document containing the details of the problem that is to be solved via the command line. An alternative to doing this is to use the PPPG user interface to generate the specification document and program source code.

The PPPG User Interface

The PPPG user interface (PPPGUI) allows the user to generate source code without having to manually create the XML specification document. This is advantageous as it removes some of the responsibility for the correctness of the specification document from the user. The remainder of this chapter serves as a walkthrough the PPPG user interface.

The PPPGUI consists of five tabbed panes. Four of these panes help to gather requirements about the nature of the problem at hand. The fifth pane displays the specification document corresponding to the user’s input.

General Information Pane

The general information tabbed pane serves as the front page for the PPPG application. When the program is started the user is presented with a screen similar to the one shown in Figure 4-1. The purpose of this pane is to collect general information concerning the problem for which the user wishes to generate code. The user is expected to enter the path to the directory in which the generated source code will be stored into the
**Target Directory** textbox. A *Browse* button is also provided to allow the user to look through the directory tree for a desired location for the source code.

**Figure 4-1** General information screen

The *Classname* textbox should contain the name of the file in which the solution should be stored. The *Units of execution* drop down box allows the user to select the number of threads desired in the solution.

**Datatypes Pane**

The datatypes tabbed window pane allows the user to specify the different datatypes that the generated program will have to handle. This window helps to capture information about the datatypes that the problem will evolve through during the solution process. It allows us to capture how the datatype of the problem to be solved will be changed from its initial representation to the datatype that the final solution. The radio buttons within the *Initial Problem Datatype* group box allow the user to pick the datatype
that the initial problem will be represented as programmatically. The user may also select a user defined type or class by selecting the *Other* radio button and entering the name of user defined class or type in the appropriate textbox. This is applicable to all three group boxes. The *Intermediate Datatype* group box contains radio buttons that allow the user to pick the datatype for the intermediate objects that the initial problem will be broken down into during execution of the generated code. The *Result Datatype* group box contains radio buttons that allow the user to specify the desired datatype for the problem solution.

![Figure 4-2 Datatypes window](image)

**Concurrency Organization Pane**

The concurrency organization pane allows the user to be able to specify the problem (for which the user wishes to generate source code) in terms of the terminal patterns of Figure 2-1.
The *Top Level Organization* group box contains three check boxes that allow the user to pick which organization best matches that of the problem for which a solution is being generated. The *Ordering Structure* group box allows for the user to specify the applicable sub ordering which best matches that of his or her problem. In the particular scenario where a pipeline processing pattern is applicable the user has the option to select the number of pipeline stages desired in the solution. The right side of the pane is a text area that displays real-time help for the user.

**Dependencies and Partitioning Pane**

The dependencies and partitioning pane allow the user to select what kind of relationships exist among tasks and also to determine where applicable the type of recursion a specific problem exhibits.
The Task Dependencies group box contains a drop down list box that allows the user to select whether the tasks in question are dependent or not. If the tasks are dependent the drop down list box within the Dependency Type group box is enabled. This drop down list box allows the user to pick whether the dependencies are separable or inseparable.

The Tree Structure Recursion group box contains a recursion type drop down list that allows the user to select whether or not the recursion exhibited by the problem is one-way or two-way.

The XML Viewer Pane

When the required information describing the problem that the user wishes to generate the source code for has been entered on each of the previous four panes, the Create XML button on the bottom left corner of the application will become enabled.
When the user clicks this button, the XML specification corresponding to the input on all the information panes will be created. The XML specification is displayed in the XML viewer pane as shown in Figure 4-5.

![XML Viewer Pane](image)

**Figure 4-5** The XML Viewer Pane

Advanced users may copy and paste the specification from the XML viewer pane into any text editor and modify for greater control. The new specification file must be supplied via the command line.

Upon creation of the XML documentation, the *Generate Source Code* button is enabled, the user may click it to create the source code in the directory specified in the *General* tabbed pane.

**Executing from the Command Line**

In order to generate source code from the command line the user must supply a valid XML specification document. The document may either be generated by hand according to the DTD described in chapter three of this thesis. Users may also generate the specification document using PPPGui and then further customize the generated
specification file using any text or XML editor. In order to run the PPPG tool from the command line the user must enter the following at the command prompt:

```
pppgcmandrun <specificationfilename>
```

The generated files will be stored in the target directory specified in the XML specification file. If a target directory is not supplied, the generated source code is placed in the current working directory.
CHAPTER 5
IMPLEMENTATION DETAILS

Chapter 4 presents an overview of PPPG from the user’s perspective. This chapter discusses the underlying details from a programmer’s perspective. Using an example problem, this chapter walks through code generation process and explains the implementation methodologies that are utilized within the process. In addition, the main modules that make up PPPG are discussed along with their respective functions.

A Walk through the Code-Generation Process

Consider the example of an application programmer who wishes to develop a parallel/concurrent mergesort procedure for use within his/her particular application. The programmer decides to use the PPPG tool. After careful analysis of the requirements for the procedure, the programmer creates the XML specification file shown in Figure 5-1.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE design.space (View Source for full doctype...)>  
- <design.space classname="mergeSort">  
  - <task.group name="Tasks">  
    - <grp.struct name="Recursive">  
      - <grp.sub.struct>  
        <task.decomposition>two way</task.decomposition>  
      </grp.sub.struct>  
    </grp.struct>  
  </task.group>  
</design.space>
```

Figure 5-1 Specification file for mergesort procedure

The tool begins its work after the XML file containing the specifications is passed as an input, either via the command line or through the GUI tool. PPPG attempts to open the specification file, if the file does not exist, an appropriate error message is generated
and the tool stops execution. However, if the specification file does exist, the file is opened and the XML specification is parsed using IBM’s XML4J XML parser to ensure the validity of the XML. The next step is to ensure that the XML specification document is well formed. This is done by validating the document against the DTD that is defined for the application. The DTD ensures that the specifications are in the format required by the application. If this is not the case a message indicating an error is generated and the tool stops execution. If the XML specification is both valid and well formed, the XML parser is used to create a DOM data structure. The DOM data structure presents the XML specification as a tree where each node contains data from the specification document. The next series of steps start with the creation of the Data capsule. The Data capsule is an application specific data structure that facilitates the transportation of data within the application. Upon creation of the data capsule object, the DOM data structure is traversed, and at each node specification information (originally from the user XML specification) is copied into the data capsule object. When the DOM data structure has been traversed and all relevant data copied to the data capsule the actual process of code generation begins. Before code generation begins a new file (name is specified in the specifications by the user) is created by PPPG. The source code will be generated into this file. The generation of code takes place as the decision tree shown in Figure 5-2 is traversed in a top down manner. The generated code at the end of the tree traversal process corresponds to an instance of one of the terminal patterns shown in the diagram. Starting at the top of the tree, PPPG must determine what kind of top-level organization is desired for concurrency. There are three major branches that can be taken from the root of the decision tree. They each correspond to the types of top-level organization
Concurrency can be organized by the ordering of task groups, decomposition of tasks, or by decomposition of data [9]. The information on which type of top-level organization was specified in the XML specifications. All specification information is now available in the data capsule object. PPPG gets the top-level information from the data capsule at run time. Source code is generated based on the type of organization the user specified. In the case of our earlier example, merge sort's concurrency can be organized in terms of task decomposition and thus the second branch is taken. Since there are no commonalities for which code can be generated at this level, boilerplate code indicating the class name and user information is generated as shown in Figure 5-3.

Figure 5-2 Decision tree
At the first level a decision must be made about how the previously selected ordering is structured. For both the second and third major branches, the ordering can be either linear or recursive. This depends on whether the tasks (branch two) or the data (branch three) is ordered linearly or recursively.

At level one after taking branch one; a decision must be made on whether the ordering is structured regularly or irregularly. PPG makes this decision based on data from the data capsule. Regular orderings are those that do not change during execution of the algorithm, while irregular orderings refer to orderings that change dynamically during algorithm execution [7].

In our example the merge sort algorithm is organized recursively by task. In this particular situation PPG does not generate code because of inadequate commonalities between recursive and linearly partitioned tasks.

On the second level, for branch one the choice (based on data capsule data) of regular ordering corresponds to the pipeline processing terminal pattern, at this point PPG generates the remainder of the code corresponding to the afore mentioned pattern. If at the same level and branch the choice of the ordering is irregular, then the code corresponding to the asynchronous decomposition pattern is generated.
The second major branch (Order by Tasks) is more involved than the other two branches. At the second level within this branch, PPPG must make either of the following choices based on the information in the data capsule. The choice of linear ordering, which indicates that the tasks are structured as linear collections, leads to the *partitioning* intermediate pattern. All programs that have the partitioning pattern as an intermediate pattern share the characteristic that the initial problem can be spilt into tasks. At this point, if linear ordering is indicated in the data capsule, PPPG generates the code for splitting the initial problem into tasks.

At the same level and branch, if the choice of recursive ordering was specified by the user, indicating that the tasks are structured recursively, we arrive at the tree intermediate pattern. At this point PPPG cannot generate any source code until further distinctions can be made. At level four within the second branch a decision must be made based on whether the tasks are independent of each other or not. If the tasks are independent PPPG arrives at the *Embarrassingly Parallel* terminal pattern and the rest of the code corresponding to this particular pattern is generated. If the tasks are dependent then PPPG must verify whether the dependencies are separable or inseparable between the tasks. If the dependencies are separable PPPG generates the source code for the *Reduction* terminal pattern. In the case that the dependencies are not separable, the source code corresponding to the *Shared Memory* pattern is generated.

At level three still within the second branch, if PPPG has an instance of the tree intermediate pattern, as is the case with the mergesort example, a decision must be made about the nature of the recursion. PPPG must decide whether the recursion in one-way or two-way. If the recursion is one-way, source code corresponding to *Balanced Tree*
pattern is generated. On the other hand if PPPG is dealing with a two-way recursion the

*Divide and Conquer* pattern is generated. In our example, the recursion is two-way and

thus code for the divide and conquer terminal pattern is generated.

```java
public Object merge ( Object [] solutions )
{
}

public boolean baseCase ( Object p )
{
}

public Object baseSolve ( Object p )
{
}
```

Figure 5-4  Remainder of code generated for *mergeSort.java* class

Since it is now known that the code being generated is an instance of the divide

and conquer pattern. The buffer is modified to make the mergeSort class extend the

DivideAndConquer class. At this point PPPG generates the source code for

*DivideAndConquer.java*. The source code generated for this class and all other code

generated is attached as appendix A.

It is important to mention that PPPG provides genericity. Genericity is a
generalization mechanism that allows a single definition to apply to a whole set of classes

with different behaviors. Genericity is the ability to parameterize modules. It is

applicable when the implementation behavior is constant but the datatype can change. If

the user had supplied a more detailed specification that included the initial, intermediate,

and final datatypes, PPPG would have generated different parameter and return types for

the methods shown in Figure 5-4. The datatypes would have changed but the

implementation behavior for each of the methods would stay the same.
At the first level within the third major branch PPPG examines the data capsule to see if the data structures are linear or recursive. If PPPG encounters a situation in which the data structures are linear then source code corresponding to the *Geometric Decomposition* pattern is generated. Otherwise, if the data structures are found to be recursive, source code corresponding to the *Reference Following* pattern is generated.

**Modules and Class Detail**

PPPG performs the following tasks.

- Validates the XML input specifications and reports errors and warnings where applicable
- Completes the specification using default settings where necessary
- Generates the solution implementation

These tasks are implemented as two consecutive and separate modules. When PPPG receives the specification in a textual form from the command line it parses the file to transform the data into the *DataCapsule* representation. If the PPPGUI is used to create the specification the information is directly inserted into a *DataCapsule* object and the parsing stage is bypassed.

PPPG consists of two main modules, the parse and package module and the code generator module. The relationships and interactions between these modules are shown in Figure 5-4.

**Parse and Package Module**

This module as its name implies is responsible for reading and validating the XML input specifications against the DTD. After validation of the specification, the input data is analyzed and packaged into a data capsule object. The data capsule object is now ready to be passed to the code generator module.
The Syntax Analyzer object implements the parse and package module. When the syntax analyzer object is instantiated, it expects an input string containing the filename of the XML input specifications. If the input file can be opened, an instance of an XML parser is created; the parser checks the input for syntax errors and then validates it against the DTD. If syntax analysis and validation are successful, the parser will create a document object model (DOM) document. The syntax analyzer completes its work by iterating through the DOM document, collecting data and storing it in a user defined data structure called the data capsule. Upon completion the data capsule containing the specifications for the program generator is passed to the generator module.

![Figure 5-4 Structure of PPPG](image-url)
Figure 5-5 Parse and package module

SyntaxAnalyzer Class Detail. The SyntaxAnalyzer class contains two very important methods. They are the `verify` and `extractData` methods. The `verify` method is responsible for validating the XML specification and initiating the creation of the DOM document. The `extractData` method extracts the data from the DOM document and populates the DataCapsule.

Datacapsule

The data capsule module is a user defined data structure used to store data from the XML input specification. The data capsule allows easier transportation of data between the two main modules. It not as bulky as the DOM document and allows easier access to the specification data. The data capsule contains fields representing the entities
and their respective attributes as defined in the DTD. For more information on the DTD please see Chapter 3.

<table>
<thead>
<tr>
<th>SyntaxAnalyzer</th>
</tr>
</thead>
<tbody>
<tr>
<td>doc:Document</td>
</tr>
<tr>
<td>errcount:int</td>
</tr>
<tr>
<td>ONE_CHILD:int</td>
</tr>
<tr>
<td>ERROR:int</td>
</tr>
<tr>
<td>+SyntaxAnalyzer</td>
</tr>
<tr>
<td>+verify:void</td>
</tr>
<tr>
<td>+extractData:DataCapsule</td>
</tr>
<tr>
<td>+main:void</td>
</tr>
</tbody>
</table>

Figure 5-6 SyntaxAnalyzer UML diagram

DataCapsule Class Detail. The DataCapsule class contains several variables to store specification information. It also provides the `generateXml` method which is responsible for transforming the current contents of the DataCapsule into its equivalent XML specification. This method is used primarily by PPPGUI.

Code Generator Module

The code generator module upon receipt of the data capsule begins the work of creating the desired output program based on the data contained data capsule object. The information in the DataCapsule usually represents a path down the tree given in Figure 5-2. The CodeGenerator object contains the implementation of the Code Generator module as described above. The code generator expects a data capsule object and a string containing the path for the destination file when it is initialized. The generation process itself involves making decisions at each level of the tree. At each non-terminal tree node, code corresponding to the particular path taken is generated and added to the output file until a terminal pattern is reached. Traversing the tree in Figure 5-2 from the root to any of the leaf nodes generates the code.
CodeGen Generator Class Detail. The CodeGenerator class contains the various methods for generating the different terminal patterns available. It also contains methods to handle generation of either C++ or Java source code.
Figure 5-8  CodeGenerator UML diagram
CHAPTER 6
GENERATED PROGRAMS

This chapter focuses on several example programs that were developed based on patterns generated by PPPG. Three sample programs are discussed. These programs illustrate the functionality and usability of PPPG and the efficiency of the generated programs. The first program is a concurrent implementation of the classic mergesort algorithm. The second is an implementation of the game of life and the final program simulates an assembly line. Each of these programs and their corresponding base patterns will be discussed in this chapter.

Concurrent Implementation of Mergesort

The sorting algorithm Mergesort produces a sorted sequence by sorting its two halves and merging them. With a time complexity of $O(n \log(n))$ mergesort is optimal. In order to generate a concurrent mergesort algorithm using PPPG a valid specification file describing the algorithm was created. For this particular instance the specification file contained information about the datatype in which the problem instance would be supplied to the program. It also contained information about the user defined structure Task that the threads would manipulate while solving the problem. The specification\(^1\) file was passed as input into PPPG and an instance of the DivideAndConquer terminal pattern was generated. The file generated is shown in Figure 6-1.

\(^1\) The complete specification file for the concurrent mergesort algorithm can be found in appendix B of this thesis paper.
The next step is to implement the abstract methods that define how the problem is to be solved. It is important to note that all the details about concurrency and thread scheduling are not visible to the programmer. These details are contained in the superclass `DivideAndConquer` generated by PPPG. The programmer need only worry about how to implement the methods provided in the generated file.

In this particular example the programmer will need to define a new class `Task`. The methods within the Mergesort.java file shown in Figure 6-1. should be implemented as follows, the `baseSolve` method is responsible for solving an atomic instance of the problem and returning the result a `Task` object. The `baseCase` method examines a `Task` object and returns a true or false value based on whether the `Task` can be divided into smaller tasks or not. The split method divides a `Task` object into smaller `Tasks` and returns and array of `Tasks`. The merge method accepts an array of `Tasks` and merges them into a single `Task`.

```
public class MergeSort extends DivideAndConquer {

    public MergeSort() {
    }

    public Task baseSolve(Task p) {
        /**@todo: implement this DivideAndConquer abstract method*/
    }

    public Task[] split(Task problem) {
        /**@todo: implement this DivideAndConquer abstract method*/
    }

    public boolean baseCase(Task p) {
        /**@todo: implement this DivideAndConquer abstract method*/
    }

    public Task merge(Task[] parml) {
        /**@todo: implement this DivideAndConquer abstract method*/
    }

    public static void main(String[] args) {
        MergeSort mergesort1 = new MergeSort();
    }
}
```
An implementation of the split method is shown in Figure 6-2. The complete implementation of the concurrent MergeSort can be found in appendix A.

```java
public Task[] split (Task problem) {
    Task[] ret_array = new Task[SPLIT_SIZE];
    // pick pivot
    int mid_point = problem.myArray.length / SPLIT_SIZE;
    int end_point = problem.myArray.length - 1;

    Integer[] temp1 = new Integer[mid_point];
    Integer[] temp2 = new Integer[end_point+1-mid_point];

    for (int i = 0; i < mid_point; i++)
        temp1[i] = problem.myArray[i];

    for (int i = mid_point; i <= end_point; i++)
        temp2[i-mid_point] = problem.myArray[i];

    ret_array[0] = new Task(temp1);
    ret_array[1] = new Task(temp2);

    return ret_array;
}
```

Figure 6-2 Implementation of the split method

Assembly Line Simulation

This example creates an assembly line simulation called the Math Factory. The math factory performs sequence of identical operations on an input number and returns the results after completion. Each unique operation represents a stage, and the stages together form a pipeline. The key here is to have the maximum number of stages executing concurrently at any given time during program execution. To accomplish this task, a specification\(^2\) file containing details about the initial problem representation, classes representing input into the factory, the number of stages and other information

\(^2\) The complete specification file for the math factory simulation can be found in appendix B of this thesis paper.
concerning the problem was created. This specification was passed as input to PPPG and an instance of the PipelineProcessing pattern was generated. A portion of the generated program is shown in Figure 6-3.

```java
import edu.ufl.cise.jcladels.pppplib.*;

public class MathFactory
{
    SharedQueue sharedQueue;
    pipeThread [] threadPool;
    Stage [] pipeline;
    public static int NUM_UP = 5;
    public static int OFFSET = 1;
    public static int NUM_STAGES = 6;
    private static long TIME_OUT = 2000; // 90 seconds

    private Barrier b;
    private volatile static int clk = 0;

    MathFactory()
    {
        // Create pipeline
        pipeline = new Stage[NUM_STAGES];
        // Create thread pool
        threadPool = new pipeThread[NUM_UP];
        sharedQueue = new SharedQueue();
        initialize();
    }

    public void complete()
    {
        /*@todo: implement this method*/
    }

    public SharedQueue split (Object problem)
    {
        SharedQueue nsq = new SharedQueue();
        try{
            // Add code to re-pack initial problem and put in queue
            /*@todo: implement this method*/
        }
    }
```

Figure 6-3  MathFactory code snippet.

In order to complete the Math factory, the programmer needs to create an implementation of the Event class that was supplied as the intermediate data type in the XML specification file. In the MathFactory class two methods will need to be implemented. The `split` method is responsible for dividing the initial problem into instances of the intermediate datatype and then placing them into a shared queue object.
This method returns a shared queue containing the objects to be worked on by my individual threads. The **complete** method should contain any operations to be performed after processing of the *Event* object. Such processing may include but is not limited to displaying the end result after all pipeline stages have been cleared.

The Math Factory class also contains a *Stage* object. This object contains all the operations corresponding to the different stages of the pipeline. A code snippet from the inner class Stage is shown in Figure 6-4.

```java
class Stage
{
    private int id;

    public Stage (int id)
    {
        this.id = id;
    }

    public synchronized void performComputation (pipeThread p)
    {
        switch (id)
        {
            case 1:
                /***@todo: implement this method***/
                // Stage 1 computation go here
                // Access thread Data by using p.my_data
                break;
            case 2:
                /***@todo: implement this method***/
                // Stage 2 computation go here
        }
    }
}
```

Figure 6-4 Inner Class *Stage* code snippet

**Implementation of the Game of Life**

The game of life is an example of cell automaton, which is any system in which rules are applied to cells and their neighbors in a regular grid. Life is played on a grid of square cells, like a chess board but extending infinitely in every direction. A cell can be live or dead. A live cell is shown by putting a marker on its square. A dead cell is shown by leaving the square empty. Each cell in the grid has a neighborhood consisting of the
eight cells in every direction including diagonals. The rules of the game of life are as follows:

For a space that contains a live cell:

- Each cell with one or no neighbors dies, as if by loneliness.
- Each cell with four or more neighbors dies, as if by overpopulation.
- Each cell with two or three neighbors survives.

For a space that is empty:

- Each cell with three neighbors becomes populated.

To continue our implementation of the game of life, we need to create a specification file describing the nature of the problem and details specific to this particular implementation. A completed specification is provided in appendix B. Based on the input to PPPG an instance of the *Embarrassingly Parallel* pattern was generated. The generated pattern is shown in Figure 6-5.

The generated file shown above shields the programmer from the details of concurrency. The programmer need only concern themselves with the details of the problem solution alone. In the game of life example the initial problem is represented in a multi dimensional array. The *split* method is responsible for dividing up portions of the array into *Task* objects which are then inserted into a shared queue. The *processResults* method will retrieve objects from the result queue and possibly update the display or perform any other similar actions. The *taskProcessing* method will process a Task object and update the result queue.

In this chapter we have examined three programs created based on patterns generated by PPPG. These patterns have enabled the programmer to focus on the task of what the program needs to do, rather than worry about details on how to make the
program run concurrently and verifying thread safety and other related issues. By using PPPG to generate concurrent/parallel versions of the program a lot of development time is saved.

```java
import edu.ufl.cise.joladele.ppglib.*;
public class Life extends EmbarrassinglyParallel
{
    void split(int [] [] problem)
    {
        /**@todo: implement this EmbarrassinglyParallel abstract method*/
        /**
         * This function should split the initial problem into smaller components
         * which can be handled by individual units of execution.
         * All the subproblems should be placed in the work queue "q"
        **/
    }

    Object processResults()
    {
        /**@todo: implement this EmbarrassinglyParallel abstract method*/
        /**
         * This function should combine all the individual results
         * contained in the "result" queue
        **/
    }

    Task taskProcessing(Task task)
    {
        /**@todo: implement this EmbarrassinglyParallel abstract method*/
    }
}
```

Figure 6-5 Game of life
CHAPTER 7
CONCLUSION

From the onset of this research project, the desired purpose was to create a program generator that enables the user to create portable and efficient parallel/concurrent programs while minimizing overall development time. This research project has been able to accomplish these goals. This chapter highlights the features of PPPG and their significance to end-users. This chapter also discusses some limitations of PPPG and corresponding suggestions for improvement.

Summary of Work Done

A program generator is very ineffective if there is no way for the user to specify what kind of programs are to be generated. Therefore, the need arose to create a specification language that can adequately and robustly communicate the details of the user’s problem in a manner that the PPPG tool can understand and generate the appropriate source code to solve the problem. The decision was made to use XML. A DTD to represent the algorithm design space of Figure 2-2. was created to help the user map their problem to one of the terminal patterns within the domain. This DTD not only helped describe the user’s problem but was used to constrain and validate input XML specifications into the PPPG tool.

The creation of the program generator involved developing several modules responsible for parsing and validation of the input specification, extracting the data from the specification, and generating the appropriate terminal pattern corresponding to the information obtained from the user. A significant amount of development time went into
the creation of the *Asynchronous Decomposition, Divide and Conquer, Embarrassingly Parallel, Pipeline Processing, Protected* and *Separable Dependencies* terminal patterns, as the patterns had to be initially created by hand before automating their creation using PPPG.

The development of the PPPG was completed with the creation of a graphical tool that simplifies the specification creation process and removes the burden of specification correctness from the user. This tool also permitted code generation directly from graphical interface.

**Future Work**

While this research effort has achieved the initial goals of simplifying the creation of parallel programs, and portability of the generated program, there are some improvements that can be made to increase the functionality and usability of the system.

Two of the terminal patterns within the algorithm structure design space are not included in this initial version of PPPG. Since this work is based on ongoing research, enough information on the *Reference Following* and *Balanced Tree* terminal patterns were not available at the time of development. As research uncovers more information about these patterns and their implementations, adding the pattern implementations to PPPG will provide completeness and allow the user to be able to generate a broader range of parallel programs.

With several of the patterns generated by PPPG, if information about the specific platform on which the generated program will used is made available, optimizations may be made to make the generated program more efficient by generating a more specialized version of the given pattern. It is however important to note that while any such change
may increase efficiency, the portability of the generated source code may be significantly affected.

An added suggestion is to include functionality that permits PPPG to generate its target programs in programming languages other than the Java language. Such functionality will give the user freedom to generate their solution in languages that they may be more familiar with. This functionality while increasing usability will add additional complexity to the generated programs and greatly decrease portability. Using the Java programming language for the generated programs provides a language with in-built multithreading and allows the JVM to handle thread scheduling issues. Not all programming languages support built-in multithreading and thus PPPG will have to use operating system function manually and may also have to worry about thread scheduling issues, thus greatly increasing the complexity of the generated programs.

Finally, while PPPG provides genericity, there are specific situations in which a user defined class may need to be cast into a primitive type or vice-versa. Such situations currently require that the programmer manually modify the generated code to ensure that these type conversions do not result in compilation errors. Improvements may be made so that this sort of occurrences can be effectively handled within the program generator.
package MergeSort;
public class MergeSort extends DivideAndConquer
{
    static int SPLIT_SIZE = 2;
    public Task[] split ( Integer [ ] problem )
    {
    }
    public Integer [ ] merge ( Task [] solutions )
    {
    }
    public boolean baseCase ( Task p )
    {
    }
    public Object baseSolve ( Task p )
    {
    }
}

public class MergeSort extends DivideAndConquer
{
    static int SPLIT_SIZE = 2;
    public Task[] split ( Task  problem )
    {
        Task [ ] ret_array = new Task [SPLIT_SIZE];
        // pick pivot
        int mid_point = problem.myArray.length / SPLIT_SIZE;
        int end_point = problem.myArray.length - 1;

        Integer [] temp1 = new Integer[mid_point];
        Integer [] temp2 = new Integer[end_point+1-mid_point];
for (int i = 0; i < mid_point; i++)
    temp1[i] = problem.myArray[i];

for (int i = mid_point; i <= end_point; i++)
    temp2[i-mid_point] = problem.myArray[i];

ret_array[0] = new Task (temp1);
ret_array[1] = new Task (temp2);

return ret_array;
}
public Task merge ( Task [] solutions )
{
    //return new Task ();
    // basic case when even
    int tot_elem = 0, ret_index = 0, first = 0, scnd = 0;
    int num_elem = solutions.length;
    for ( int i = 0; i < num_elem;++i )
        tot_elem += solutions[i].size;

    Integer [] ret = new Integer[tot_elem];

    while ( first < solutions[0].myArray.length ||
            scnd < solutions[1].myArray.length )
    {
        if ( first < solutions[0].myArray.length &&
             scnd < solutions[1].myArray.length)
        {
            if (solutions[0].myArray[first].intValue() <
                solutions[1].myArray[scnd].intValue())
                ret[ret_index++] = solutions[0].myArray[first++];
            else
                ret[ret_index++] = solutions[1].myArray[scnd++];
        }
        else
        {
            if (scnd == solutions[1].myArray.length) // sec array finished
                ret[ret_index++] = solutions[0].myArray[first++];
            else if ( first == solutions[0].myArray.length)
                ret[ret_index++] = solutions[1].myArray[scnd++];
        }
    }
    return new Task (ret);
}
public boolean baseCase ( Task p )
{
if (p.myArray.length == 1)
    return true;
else
    return false;
}
public Task baseSolve ( Task p )
{
    return p;
}
public static void main(String [] args )
{
    Task probtask;
    if ( args.length != 0){
        Integer [] numbers  = new Integer [args.length];
        for ( int i = 0; i < args.length; i++ )
            numbers[i] = new Integer ( args[i]);
        probtask = new Task(numbers);
    }
    else
    {
        Integer [] prob = {new Integer(13), new Integer(6), new Integer(4),
            new Integer(14), new Integer(3), new Integer(12),
            new Integer(11), new Integer(7), new Integer(15),
            new Integer(9), new Integer(16), new Integer(5),
            new Integer(2), new Integer(1), new Integer(8),
            new Integer(10)};
        probtask = new Task (prob);
    }
    MergeSort x = new MergeSort();
    Object y = x.solve (probtask);
    System.out.print(y);
}
}

Source Listing for DivideandConquer.java

import edu.ufl.cise.joladele.cppj.*;
public abstract class DivideAndConquer {
    public  abstract Task[] split ( Task  problem );
    public  abstract Task merge  ( Task [] solutions );
    public  abstract boolean baseCase ( Task p );
    public  abstract Task baseSolve ( Task p );
    public Object solve ( Task  p )
    {
        if ( baseCase (p))
            return baseSolve(p);
        else
        {

}}
Task [] subproblems = split (p);
int N = subproblems.length;
Task [] subsolutions = new Task [N];
Barrier b = new Barrier(N + 1);

// start threads to solve subproblems
for ( int i = 0; i < N; i++ )
    (new solverThread(b, i, subproblems, subsolutions)).start();

// wait for all subsolutions to be completed
try {  b.barrier(); }
catch ( InterruptedException e )
{
    ;
}

// merge all subsolutions and return result
return merge ( subsolutions );

class solverThread extends Thread {

    Barrier passed_barrier;
    int number;
    Task [] subproblems;
    Task [] subsolution;

    public solverThread()
    {
        passed_barrier = null;
    }

    public solverThread(Barrier b, int num, Task [] subp, Task [] subsol)
    {
        passed_barrier = b;
        number = num;
        subproblems = subp;
        subsolution = subsol;
    }

    public void run ()
    {
        subsolution[number] = (Task)solve(subproblems[number]);
        try
        {
           // passed_barrier.barrier();
        }
        catch ( InterruptedException e )
APPENDIX B
SPECIFICATIONS FOR EXAMPLE PROGRAMS

Specification for MergeSort Example

```xml
<?xml version = "1.0" encoding = "UTF-8"?>
<!DOCTYPE design.space SYSTEM "http://www.cise.ufl.edu/~joladele/PPPG/taskDecomposition.dtd">
<design.space classname = "MergeSort" sourcepath = "C:\Documents and Settings\joladele\My Documents\My Work" initdatatype = "Integer [ ]" units = "6">
  <task.group name = "Tasks" datatype = "Task">
    <grp.struct name = "Recursive">
      <grp.sub.struct>
        <task.decomposition>two way</task.decomposition>
      </grp.sub.struct>
    </grp.struct>
  </task.group>
</design.space>.
```

Specification for Assembly Line Simulation

```xml
<?xml version = "1.0" encoding = "UTF-8"?>
<!DOCTYPE design.space SYSTEM "http://www.cise.ufl.edu/~joladele/PPPG/taskDecomposition.dtd">
<design.space classname = "MathFactory" sourcepath = "C:\Documents and Settings\joladele\My Documents\My Work" initdatatype = "Object" units = "5">
  <task.group name = "Ordering" datatype = "Event">
    <grp.struct name = "Regular" numstages = "6"/>
  </task.group>
</design.space>
```

Specification for Game of Life

```xml
<?xml version = "1.0" encoding = "UTF-8"?>
<!DOCTYPE design.space SYSTEM "http://www.cise.ufl.edu/~joladele/PPPG/taskDecomposition.dtd">
<design.space classname = "Life" sourcepath = "C:\Documents and Settings\joladele\My Documents\My Work" initdatatype = "int [ ] [ ]" units = "5">
  <task.group name = "Tasks" datatype = "Task">
    <grp.struct name = "Linear">
```

56
<grp.sub.struct>
  <task.relationships type = "Independent"/>
</grp.sub.struct>
</grp.struct>
</task.group>
</design.space>
LIST OF REFERENCES


http://charm.cs.uiuc.edu/ppl_research/ldbal/

BIOGRAPHICAL SKETCH

Jean-David Oladele was born June 25, 1976 in Lyon, France. He received his Bachelor of Engineering degree in computer engineering from the University of Florida, Gainesville, Florida in May 1999. He worked as a junior consultant for Software Architects Inc. in Tampa, Florida, for about a year and a half. In the fall of 2000, he began graduate study in the Department of Computer and Information Science and Engineering at the University of Florida for the Master of Engineering degree in computer engineering. He received the degree in December 2002.