Automated Adaptive Software Maintenance: A Methodology and Its Applications

Wesley S. Tansey

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science and Applications

Dr. Eli Tilevich, Chair

Dr. Godmar V. Back

Dr. Calvin J. Ribbens

May 22, 2008

Blacksburg, Virginia

Keywords: Software Maintenance, Adaptive Maintenance, Program Synthesis, HPC, Marshaling, Upgrading, Frameworks, Metadata

Automated Adaptive Software Maintenance: A Methodology and Its Applications

Wesley S. Tansey

ABSTRACT

In modern software development, maintenance accounts for the majority of the total cost and effort in a software project. Especially burdensome are those tasks which require applying a new technology in order to adapt an application to changed requirements or a different environment.

This research explores methodologies, techniques, and approaches for automating such adaptive maintenance tasks. By combining high-level specifications and generative techniques, a new methodology shapes the design of approaches to automating adaptive maintenance tasks in the application domains of high performance computing (HPC) and enterprise software. Despite the vast differences of these domains and their respective requirements, each approach is shown to be effective at alleviating their adaptive maintenance burden.

This thesis proves that it is possible to effectively automate tedious and error-prone adaptive maintenance tasks in a diverse set of domains by exploiting high-level specifications to synthesize specialized low-level code. The specific contributions of this thesis are as follows: (1) a common methodology for designing automated approaches to adaptive maintenance, (2) a novel approach to automating the generation of efficient marshaling logic for HPC applications from a high-level visual model, and (3) a novel approach to automatically upgrading legacy enterprise applications to use annotation-based frameworks.

The technical contributions of this thesis have been realized in two software tools for automated adaptive maintenance: MPI Serializer, a marshaling logic generator for MPI applications, and Rosemari, an inference and transformation engine for upgrading enterprise

applications.

This thesis is based on research papers accepted to IPDPS '08 [93] and OOPSLA '08 [92].

Acknowledgments

No grad student is an island, and I am surely no exception. My friends, family, and colleagues have ameliorated my life and I truly appreciate the kindness they all have shown me. I would like to take this opportunity to thank many of those who have aided me in one way or another along the way.

The never-ending support and love of my parents has sustained, comforted, and driven me throughout my life. Any accomplishments that I have achieved in my time here at Virginia Tech would not have been possible if I did not have them by my side. They are the most caring, loving, and genuine people I have ever known. The faith that both my brother and sister have had in me spurred me on through the most difficult challenges.

For the five years that I have known her, Lindsay La Forge has been the most influential person in my life. She has always sincerely cared about me and never allowed me to slip down the wrong path. The degree to which she has driven me to succeed is immeasurable, and it seems as though every day I find myself aspiring to possess her strength of will. She is the most precious thing in my life, and I love her tremendously.

The entire La Forge family has always made me feel at home. I could not imagine a more delightful, hospitable, or encouraging person than Jeannie La Forge, who cared for me, worried about me, and fed me like I was her own son. The tenacity of Lola La Forge has

served as a constant reminder that I should never waver from my goals, even when the largest challenges bear down on me. Mike, Jessica, and Lainey La Forge all welcomed me into their home and made every visit a pleasure.

My friend and soon-to-be boss, David Fogel, has given me guidance for many years. While still in high school, his book *Blondie24* inspired me to become a Computer Scientist; later, his enthusiasm in my undergraduate research sparked my ambition and strengthened my desire to enter a graduate program; most recently, he has shown confidence in me by hiring me as a professional researcher. Whether about research, employment, or life, each discussion I had with him left me feeling reassured that I was on the right track.

Having received both my undergraduate and graduate education from Virginia Tech, I owe a great debt of thanks for the continuous support and resources provided to me by the Computer Science department's faculty and staff. Libby Bradford and Terry Arthur were kind enough to let me act as a representative example of a computer science student. Without the chance to reminisce with incoming freshmen about all of the great experiences I've had as a CS student at Virginia Tech, I would not have grown to love this school or department as much as I do today.

Dr. Cal Ribbens was able to melt away my stress whenever an issue with the department or my research arose. His help and guidance on high performance issues was critical to the success of the first half of my thesis.

My advisor, Dr. Eli Tilevich, has taught me several invaluable lessons throughout my time as a graduate student. While many of these lessons focused on how to perform top-notch research, others were less conventional. In particular, I learned that if a tight paper deadline prevents me from going to the gym, the antioxidants in a bag of dark chocolate are a sufficient substitute. Perhaps most importantly, I learned that if I choose to enter into a PhD program,

it would be an illogical and irrational decision– and the right one.

The precise insights offered by Dr. Godmar Back were priceless. His talent for revealing deficiencies in an approach cost me countless nights of rest, as each project, paper, or presentation invariably needed to be reworked. Nevertheless, addressing his comments always made sure the work would be tempered against even the toughest criticism.

My labmates have been vital to my completion of graduate school and I would like to thank them all for providing a shoulder to lean on. Many of my research projects could not have been completed without Cody Henthorne's seemingly endless patience in answering my Eclipse questions and listening to my off-the-wall ideas. Every time frustration or stress reached a boiling point, Andrew Hall was willing to listen to my cathartic rants, no matter how incoherent. Conversations with Sriram Gopal helped to solve several research challenges, and his determination and dedication to a project are something I truly respect. Jeremy Archuleta provided me with indispensable advice on my first project, paper, and presentation, making my introduction to research as painless as possible.

My friends outside of the research world have kept me in touch with reality and allowed me to keep my struggles in perspective. Eddie Sumbry was always available for a cathartic session of competition, each of which invariably concluded with us both screaming in jest, smiling, and feeling relaxed. The conversations and meals I had with Ali Sohanghpurwala were a highlight of my time in Blacksburg. Discussions with Neil Mathews have never failed to boost my spirits and increase my ambition; he is one of the most reliable and supportive friends I have ever had.

Finally, I would like to thank my employer, Lincoln Vale, for giving me the opportunity to apply my research capabilities in the real world.

Contents

1 Introduction **1**

1.1 The Case for Automated Adaptive Maintenance 1

1.2 Applicability of Automated Adaptive Maintenance 3

1.2.1 High Performance Computing 3

1.2.2 Enterprise Software 4

1.3 A Common Methodology 5

1.3.1 MPI Serializer 6

1.3.2 Rosemari 6

1.4 Thesis Statement 7

1.5 Contributions 7

1.6 Overview of Thesis 8

2 MPI Serializer **9**

2.1 Introduction 9

2.2 Motivation . . . 11
2.2.1 Design Objectives . . . 11
2.2.2 Prior Approaches . . . 12
2.2.3 Motivating Example . . . 14
2.3 A Generalized Automatic C++ MPI Marshaling Approach . . . 16
2.3.1 User Interface . . . 18
2.3.2 Handling C++ Language Features . . . 19
2.3.3 Generating Efficient Marshaling Code . . . 23
2.3.4 Using Generated Code . . . 24
2.4 Case Studies and Performance Results . . . 26
2.4.1 Micro-benchmarks . . . 28
2.4.2 Parallelizing NEAT . . . 29
2.4.3 mpiBLAST 2.0 . . . 31
2.5 Future Work . . . 33
2.6 Conclusions . . . 33
3 Rosemari **35**
3.1 Introduction . . . 35
3.2 Motivating Example . . . 36
3.3 Approach Overview . . . 39

3.3.1 Representative Examples 40
3.3.2 Upgrade Patterns 42
3.3.3 Transformation Rules 44
3.4 Inference Algorithm 45
3.4.1 Decomposing Representative Examples 45
3.4.2 Component Inference Algorithms 46
3.4.3 Merging Algorithms 51
3.5 Evaluation 53
3.5.1 Inferred Refactorings 53
3.5.2 Case Studies 59
3.6 Future Work 59
3.7 Conclusions 60

4 Related Work **70**

4.1 Introduction 70
4.2 MPI Serializer 70
4.3 Rosemari 73
4.3.1 Technique Classification 73
4.3.2 Program Transformation Systems 74
4.3.3 Program Differencing 75

4.3.4 API Evolution 75

4.4 Programming by Demonstration 76

5 Thesis Contributions and Conclusions 78

5.1 Thesis Contributions 78

5.2 Conclusions 79

Bibliography 81

A MPI Serializer Code Examples 95

A.1 Master to Worker Communication 95

A.1.1 Header File 95

A.1.2 CPP File 97

A.2 Worker to Master Communication 101

A.2.1 Header File 101

A.2.2 CPP File 102

List of Figures

1.1 The taxonomy of types of software maintenance, as specified by the ISO/IEC 14764 standard. 2

2.1 An overview of the control flow for MPI Serializer. 17

2.2 Selecting a subset of an object graph in MPI Serializer. 19

2.3 Accessing non-public fields using our C++ standard-compliant strategy. . . . 22

2.4 An example of mapping bounded subsets of an object graph to `MPI_Datatype`'s. 25

2.5 Benchmark results comparing the total marshaling and unmarshaling time required for the `Mass` class example. 28

2.6 Benchmark results comparing the marshaling time required for a population of organisms in NEAT. 30

2.7 Benchmark results comparing the marshaling time required for a collection of genome alignment results in mpiBLAST. 32

3.1 Comparisons of a JUnit test case in version 3 and 4. 37

3.2 Comparison of a JUnit test suite in version 3 and 4. 38

3.3 The Rosemari context menu. 40

3.4 An example showing the ambiguity inherent in inferring refactorings between two arbitrary representative examples. 41

3.5 The generated transformation rules to refactor a JUnit 3 test case class to use JUnit 4. 61

3.6 A simple Enterprise Java Bean example in EJB 2 and 3. 62

3.7 The levels at which each representative example is examined. 62

3.8 Upgrade patterns as partial orders. 62

3.9 The level decomposition of the EJB example. 63

3.10 The *LevelRestructurings* algorithm for calculating the required set of restructurings to transform between two levels. 63

3.11 The *LevelConstraints* algorithm for calculating the context-free set of constraints for a level. 64

3.12 The *NamingConvention* algorithm for generalizing a set of names to a naming convention. 64

3.13 The *ContextConstraints* algorithm for calculating the additional constraints for every level. 65

3.14 The *SalienceValues* algorithm for calculating the order of execution for every level. 66

3.15 The *LevelTransformations* algorithm for generating a set of transformation rules for a level. 67

3.16 The *ProgramTransformations* algorithm for inferring a set of transformation rules from two representative examples. 68

3.17 The rules learned to transform the EJB example from version 2 to version 3. 69

List of Tables

2.1 Comparison of MPI Serializer (MPI S.) to directly related state-of-the-art approaches 14

3.1 A summary of the requirements for application elements using JUnit versions 3 and 4. 38

3.2 The seven different upgrading scenarios and their corresponding upgrade patterns. 56

3.3 The accuracy of the inferred rules for the seven different upgrading scenarios. 57

3.4 The accuracy of the inferred rules for the seven different upgrading scenarios. 57

3.5 Upgrade statistics for four real-world case studies. 59

Chapter 1

Introduction

1.1 The Case for Automated Adaptive Maintenance

After a software application is released, the primary focus of developer efforts shifts from implementing required features to maintaining the existing code base. According to the ISO/IEC 14764 standard [49], developer actions in this phase, known as *software maintenance*, fall into one of four categories: corrective, preventative, adaptive, or perfective. Corrective actions correct problems discovered after delivery. Preventative actions correct latent defects before they result in operational failures. Adaptive actions enable execution in a different environment from the original target environment. Perfective actions improve the maintainability or performance of the application. Corrective and preventative maintenance are grouped more generally as corrections, while adaptive and perfective maintenance are considered enhancements. Figure 1.1 shows the taxonomy of software maintenance activities.

Software maintenance comprises the largest portion of the total cost and effort of the software development process [11, 57], with enhancements constituting the majority of maintenance

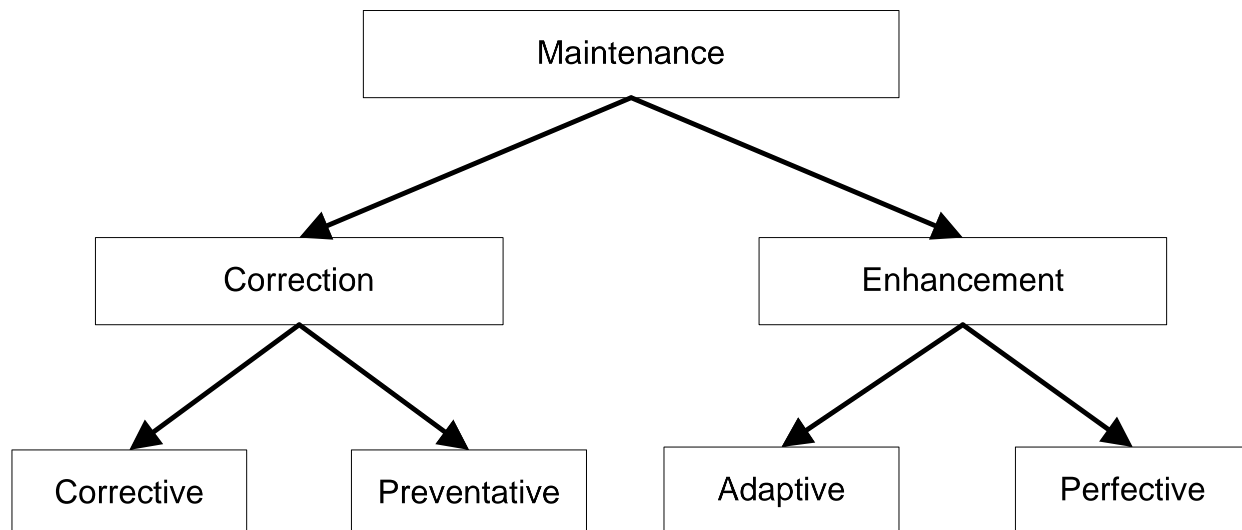

Figure 1.1: The taxonomy of types of software maintenance, as specified by the ISO/IEC 14764 standard.

tasks [61, 105]. There are many contributing factors for this: the complexity of software grows exponentially in relation to the size of a program [9], maintenance programmers are often not the original developers of the application, and software documentation may be inadequate or non-existent. Unsurprisingly, addressing changed requirements or fixing program defects requires an effort that is directly proportional to the size of a program [43].

In addition to the costly complications generally associated with software maintenance, adaptive maintenance includes its own specific challenges. By definition, adaptive maintenance tasks require the developer to possess a set of skills beyond those necessary to produce the original release. Often, the developer must learn to use different APIs or language features in order to migrate an application to a new middleware platform or framework vendor. As such, while the actual migration task may be trivial to perform once the developer is familiar with the new technology, achieving such familiarity is often time-consuming and expensive. The cost of this training period contributes to the task's total overhead and provides strong motivation for tools and approaches which can automate adaptive maintenance.

1.2 Applicability of Automated Adaptive Maintenance

The potential benefit of automated adaptive maintenance tools is not limited to a single domain, but rather spans a broad spectrum of modern software development. This thesis illustrates the broad necessity of such tools by exploring two vastly different areas: high performance computing (HPC) and enterprise software. The primary concern of HPC developers is to produce highly efficient and optimized code, capable of solving intense scientific and engineering problems in a minimal amount of time. Conversely, enterprise developers are typically concerned with the software engineering quality of their solution, and place a high premium on elegant designs. Despite the apparent dichotomy between these two domains, developer productivity in both can be improved through automated adaptive maintenance.

1.2.1 High Performance Computing

Recent studies [47, 48] have indicated that Shared Memory Multiprocessor (SMMP) programming models can lead to higher programmer productivity compared to Distributed Memory Multiple Processor (DMMP) models (i.e., compute clusters). A key difference between these two programming models is the need to pass program state between multiple processors across the network in DMMP. The programming technique for accomplishing this goal is called *marshaling*, also known as *serialization*. The technique entails representing the internal state of a program in an external format. Data in an external format can then be passed across the network. The reverse of this technique is called *unmarshaling*, also known as *deserialization*. These techniques are tedious and error-prone to implement by hand without library or automated tool support and likely contribute to the discrepancies in programmer productivity between the SMMP and DMMP models.

Nevertheless, the relatively lower cost of DMMP hardware makes it more common and

accessible to a wider group of parallel programmers [96]. The primary middleware facility for parallel programming on a compute cluster is the Message Passing Interface (MPI) [68]. MPI is ubiquitous, being available on virtually every parallel cluster-based computing system and has a standardized application programming interface (API). In addition, MPI aims at providing a uniform API for multiple programming languages such as Fortran, C, and C++. Consequently, this API is tailored to the lowest common denominator, providing only low-level marshaling facilities. As a result, C++ programmers are forced to provide tedious and error-prone code to interface their C++ application code with the MPI marshaling facilities. Therefore, automated tools for mapping higher-level features of C++ to lower-level marshaling facilities of MPI have the potential to significantly improve programmer productivity.

1.2.2 Enterprise Software

By providing reusable designs and a predefined architecture, frameworks enable developers to streamline the software construction process and have consequently become a mainstay of enterprise software development. One design decision that has to be made when creating a framework is how application and framework objects will interact with each other. Traditionally, frameworks have employed type and naming conventions to which the programmer must adhere. However, since metadata support has been added to modern object-oriented languages (e.g., Java Annotations and .NET Attributes), frameworks have increasingly moved to using language-supported metadata facilities. Being embedded within the source code next to the program elements they describe, annotations provide a more concise and robust approach to providing declarative information. Predictably, many existing frameworks have switched from using type and naming conventions to using annotated Plain Old Java Objects (POJOs) in their latest releases.

The benefits of upgrading to the latest annotation-based version of a framework may not be worth the programming effort required to apply these refactorings manually, resulting in an anti-pattern that we call *Version Lock-in*. Also, as noted in a recent article [78] describing metadata-based enterprise frameworks, annotations present an additional challenge: the tight coupling between annotations and source code may render switching between different framework vendors prohibitively expensive for large software projects. Such a high re-engineering cost results in the phenomenon known as the *Vendor Lock-In* anti-pattern [18]. The development of an automated tool for upgrading framework-based applications suffering from these hindrances would enable enterprise developers to reap the benefits of upgrading without having to manually change their code base.

1.3 A Common Methodology

At first glance, the two domains discussed above and their specific adaptive maintenance challenges bear little resemblance to each other. Nevertheless, a common methodology can be used to design automated approaches for both tasks. This methodology requires that the developer adheres to three pillars: automation, high-level specifications, and low-level code synthesis. The pillars are described in detail below:

1. *Automation* - Modern compilers and integrated development environments (IDEs) provide third-party tools with many facilities, such as abstract syntax tree (AST) information and source manipulation routines. The first pillar integrates these facilities to initialize and apply the approach automatically.

2. *High-Level Specifications* - The benefits gained by automation may be outweighed if the automated approach as too steep of a learning curve. The second pillar aims at

lowering the barrier to entry by enabling the user to express their intent via intuitive abstractions.

3. *Low-Level Code Synthesis* - The tedious and error-prone nature of many adaptive maintenance tasks makes them costly. The third pillar streamlines and simplifies the task by generating code which would otherwise have to be written by hand.

This methodology shaped the design of the MPI Serializer and Rosemari approaches, described next.

1.3.1 MPI Serializer

Our novel approach, MPI Serializer, overcomes the challenges of mapping higher-level features of C++ to the lower-level marshaling facilities of MPI. The design of MPI Serializer adheres to the three pillars of our methodology as follows:

1. *Automation* - A compiler tool enables our approach to automatically parse a C++ application and create a model of the class hierarchy.

2. *High-Level Specifications* - Marshaling logic is specified via a visual model that provides graphical abstractions for a wide variety of C++ language features.

3. *Low-Level Code Synthesis* - Specialized object graph traversals and standard MPI marshaling calls are generated based on the visual specification.

1.3.2 Rosemari

Our novel approach, reified in the Rosemari tool, solves both the Version and Vendor Lock-In problems associated with annotation-based frameworks. The design of Rosemari adheres to

the three pillars of our methodology as follows:

1. *Automation* - AST and source manipulation libraries provided by the IDE automate the parsing and transformation of legacy application classes.

2. *High-Level Specifications* - Upgrade rules are inferred from "before-and-after" examples that developers have likely already created for tutorials. These rules are then expressed using our domain-specific language (DSL).

3. *Low-Level Code Synthesis* - Conditional and transformation DSL statements are generated by our inference algorithm.

1.4 Thesis Statement

This thesis asserts the following statement:

> *It is possible to effectively automate tedious and error-prone adaptive maintenance tasks in a diverse set of domains by exploiting high-level specifications to synthesize specialized low-level code.*

This research proves this thesis by illustrating adaptive maintenance challenges in two vastly different areas, HPC and enterprise development, and following a common methodology to develop software tools to automate these tasks.

1.5 Contributions

This thesis makes the following contributions:

1. A common methodology for designing automated approaches to adaptive maintenance.

2. A novel approach to automating the generation of efficient marshaling logic for HPC applications from a high-level visual model.

3. A novel approach to automatically upgrading legacy enterprise applications to use annotation-based frameworks.

1.6 Overview of Thesis

This thesis is based on two previous publications [93, 92] and the remainder is organized as follows. Chapters 2 and 3 detail the motivation, design, and implementation of MPI Serializer and Rosemari, respectively. Chapter 4 discusses related work. Chapter 5 enumerates the contributions of this thesis and presents concluding remarks.

Chapter 2

MPI Serializer

2.1 Introduction

Parallel programming for High Performance Computing (HPC) remains one of the most challenging application domains in modern software engineering. At the same time, the programmers who create applications in this complex domain are often non-experts in computing. The primary users of parallel processing tend to be lab scientists and engineers, many of whom have limited experience with parallel programming. The dichotomy between the intrinsic difficulty of this application domain and the non-expert status of its many application developers is a major bottleneck in the scientific discovery process. This provides a strong motivation for research efforts aimed at making parallel processing more intuitive and less error-prone.

C++ is one of the foremost higher-level programming languages that provides support for object oriented, procedural, and generic programming models. While large portions of engineering and scientific code have been written in Fortran, C++ is often a preferred language

for programming complex, performance-conscious models due to its availability, portability, efficiency, and generality [6, 67, 99]. As C++ is being used more and more in scientific and engineering computing, the importance of alleviating the burden of implementing marshaling logic in C++ for MPI grows due to the number of non-expert users in these domains. However, marshaling C++ data structures is non-trivial due to the inherent complexity of the language.

This chapter presents a novel approach to overcoming the challenges of mapping higher-level features of C++ to the lower-level marshaling facilities of MPI. Our approach eliminates the need for the programmer to write any marshaling code by hand. Instead, our tool uses compiler technology to parse C++ code and to obtain an accurate description of a program's data structures. It then displays the resulting description in a graphical editor that enables the programmer to simply "check-off" the subset of a class's state to be marshaled and unmarshaled. Our tool then automatically generates efficient MPI marshaling code, whose performance is comparable to that of hand written code.

Our approach provides a more comprehensive level of support for C++ language features than the existing state of the art [77, 50, 42, 46]. Specifically, we can marshal effectively and efficiently non-trivial language constructs such as polymorphic pointers, dynamically allocated arrays, non-public member fields, inherited members, and STL container classes. A distinguishing characteristic of our approach is that it provides support for a significant subset of the C++ language without requiring any modifications to the existing C++ source code. This makes our approach equally applicable to third party libraries as well as to programmer-written code. Our tool generates code that uses standard MPI calls and requires no additional libraries, thereby simplifying deployment.

2.2 Motivation

Automated marshaling of C++ objects for the HPC domain presents a unique set of challenges which have not been fully addressed by prior approaches. We first explain the unique challenges of automatically generating C++ marshaling functionality. Then we present a concrete example of a C++ class and briefly discuss why existing approaches are insufficient to enhance this class with efficient marshaling functionality. In the process, we also pinpoint the additional capabilities required to accomplish this goal.

2.2.1 Design Objectives

In creating an automated tool for generating marshaling functionality for C++ data structures, we set the following objectives:

1. The tool should provide abstractions to automate the handling of low-level marshaling details, to make our approach appealing to experts and non-experts alike.

2. The generated marshaling code should be highly efficient, as performance is a crucial requirement for HPC applications.

3. The tool should be able to marshal any subset of an object's state to minimize the amount of network traffic and enable multiple marshaling strategies per object rather than providing only one strategy per type (i.e., C++ class) to allow maximum flexibility.

4. The generated marshaling functionality should not require any modification to the existing code, to make our approach work with third-party libraries.

5. The generated code should use only standard MPI calls and not require any additional runtime library to ease deployment and to ensure cross-platform compatibility.

6. The marshaling technique should be able to support a subset of the C++ language that contains commonly used features including:

   (a) Non-primitive fields

   (b) Pointers

   (c) Non-public fields (i.e., private and protected)

   (d) Static and dynamic arrays

   (e) Inheritance

   (f) Standard Template Library (STL) containers

In the following discussion of previous approaches, we will refer to the above six objectives to motivate our approach and to explain why existing approaches are insufficient.

2.2.2 Prior Approaches

Several prior approaches have attempted to provide automatic serialization of C/C++ data structures for MPI applications. These approaches include both automatic tools and special purpose libraries. The automatic tools that we consider closely related work include AutoMap/AutoLink [42], C++2MPI [46], and MPI Pre-Processor [77]. In addition, Boost.MPI and Boost.Serialization [50] provide library support for seamless marshaling and unmarshaling. Next we describe and compare the features of each of these approaches to motivate our approach and demonstrate how it improves on existing state-of-the-art.

AutoMap/AutoLink [42] provides automatic marshaling of C structures by enabling the user to annotate fields to be marshaled. While the annotations provide a level of abstraction,

its granularity might be too low-level particularly for non-expert users, possibly violating objective one. Obviously, this approach will fail if the fields are a part of a structure in a third-party un-modifiable library, which violates objective five. In addition, AutoMap/AutoLink does not support C++, which violates objective six.

C++2MPI [46] and MPI Pre-Processor [77] provide automatic creation of `MPI_Datatype`'s for C/C++ data structures. An `MPI_Datatype` is a list of memory offsets describing the data to be marshaled given the base offset of a structure. However, the approach makes an implicit assumption that all memory in a structure has been allocated statically, which violates objective six. In addition, the tools do not support marshaling subsets of an object state, which violates objective three.

Finally, the Boost.MPI and the Boost.Serialization libraries [50] aim at modernizing the C++ interface to MPI by utilizing advanced generic programming techniques [28]. These libraries provide support for automatic marshaling of primitive data types, user-defined classes, and STL container classes. In order for a user-defined class to use the services of the Boost libraries, it has to supply a `serialize` method either as a class member or as an external method. To use a member method requires changes to the original source code of the class, which violates objective four. This violation can be avoided by supplying `serialize` as an external method. However, in this case `serialize` would not be able to access non-public fields of a class, without the class declaring the method as `friend`, which violates objective six. A `friend` declaration, of course, would once again violate objective four. Moreover, the Boost libraries follow the per-type marshaling strategy. That is, all objects of a given C++ class are marshaled based on the functionality of the corresponding `serialize` method, violating objective three. While the logic inside such a `serialize` method might marshal objects differently depending on some of their properties (e.g., field values), this offers only limited flexibility and might incur significant performance overheads, violating objective two.

Objectives	AM/AL	C++2MPI	MPIPP	Boost	MPI S.
1. High-level abstractions	+/-	+	+	+	+
2. Efficient Code	+	+	+	+/-	+
3. Multiple partial object marshaling	-	-	-	-	+
4. No source modification required	-	-	+	-	+
5. No run-time library required	+	+	+	-	+
6. Support C++	-	+	-	+	+
a. Non-primitive fields	-	-	-	+	+
b. Pointers	-	-	-	+	+
c. Non-public fields	-	+	-	+	+
d. Static and dynamic arrays	-	+/-	-	+	+
e. Inheritance	-	-	-	+	+
f. STL containers	-	-	-	+	+

Table 2.1: Comparison of MPI Serializer (MPI S.) to directly related state-of-the-art approaches

Table 2.1 illustrates how well each directly related approach satisfies our stated objectives, thereby motivating our approach.

2.2.3 Motivating Example

Our example comes from a well-known computational problem in astrophysics, the *N-Body* problem [58]. This example belongs to an important class of parallel algorithms utilizing long range data interactions. The algorithm computes the gravitational force exerted on a single mass based upon its surrounding masses. To express this problem in C++, a programmer might create a class `Mass` as follows:

```
class Mass {
private:
  vector<Mass> surrounding; //nearby masses
protected:
  float force_x, force_y; // resulting force
```

```
public:
    float mass; //mass value
    float x, y; //position
    ...
};
```

As part of the algorithm, we need to send an object of this class from one process to another. Furthermore, only a subset of the object's state is needed at any given time. Specifically, when sending the `Mass` object from the Master process to the Worker process, only fields `mass`, `x`, `y`, and `surrounding` are required. Conversely, when sending the object in the opposite direction, only fields `force_x` and `force_y` are needed, as the rest of the object is still in memory on the master.

Let us consider how these simple marshaling tasks can be successfully accomplished using the existing state-of-the-art approaches to C++ marshaling for MPI. In particular, we evaluate the four most directly related approaches: AutoMap/AutoLink, C++2MPI, MPIPP, and Boost.

The marshaling functionality required for sending the object from the Master to the Worker process cannot be accomplished by using any of the first three approaches, as they provide no support for marshaling fields of dynamically-determined size such as STL containers. Additionally, because these three approaches do not support partial object marshaling, they could only provide marshaling functionality for sending the object in the opposite direction by defining an intermediate structure and copying the `force_x` and `force_y` fields to a temporary instance of this structure.

The Boost libraries do provide support for the required marshaling functionality, but this support is not adequate to address all of the requirements. The programmer could add a

serialize member method to the `Mass` class to marshal the required fields for the Master to Worker communication. Notice that this approach would fail if class `Mass` could not be modified (e.g., being a part of a third-party library). However, the marshaling functionality implemented by such a serialize method would be per-type. As a result, it would be non-trivial to use different marshaling logic when sending the object back from the Worker to the Master. Recall that in this case we only want to send back the fields `force_x` and `force_y`. One way to enable such custom marshaling would be to add an extra `boolean` direction field to the class and to set it to an appropriate value before marshaling the object. Nevertheless, even if code modifications were possible, this solution might not be acceptable, as it incurs performance overheads by preventing Boost from optimizing the marshaling functionality through `MPI_Datatype`'s.

To summarize, the primary reasons why the existing state-of-the-art approaches fell short of meeting the demands of this marshaling task are that they either failed to provide adequate support for common C++ language features (i.e., non-public fields, STL container) or required extensive code modification and restructuring. Thus, we feel that this simple example sufficiently motivates the need for a better approach to automatic C++ MPI marshaling. While this was only a simple example, the case studies that we present in Section 6 further justify the need for our new approach and automatic tool.

2.3 A Generalized Automatic C++ MPI Marshaling Approach

The domain of HPC applications is computationally intensive. Despite utilizing parallel hardware resources, HPC applications often run for prolonged periods of time before arriv-

ing at a solution. Lab scientists and engineers, who are the primary developers of HPC applications, care first and foremost about decreasing this “time-to-answer.”

This presents a unique set of challenges to software engineering researchers who aim at providing novel programming tools in support of HPC application developers. These tools must provide a high degree of usability while still producing highly efficient code. These two goals are often irreconcilable. Ensuring good usability entails providing abstractions, which are commonly detrimental to performance. In order to create a user-friendly automated tool for high performance applications, we make the following design assumptions:

- Any changes to the marshaling functionality will be made via the GUI of our tool, which will re-generate the marshaling code.
- The generated marshaling code will not be modified by hand.
- The generated marshaling code places a higher priority on performance than readability.

We believe that these assumptions are reasonable, as the primary focus for HPC developers is on performance, usability, and time-to-answer.

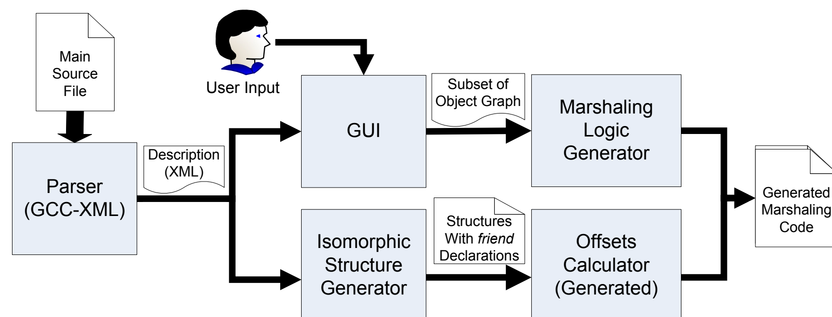

Figure 2.1: An overview of the control flow for MPI Serializer.

Next we describe the main steps of our approach in turn. Figure 2.1 shows an overview of the control flow of our approach and automated tool. The programmer interacts with the

tool through a GUI. As the first step, the tool parses the C++ data structures (e.g., classes and structs) for which marshaling code can be generated. Then the programmer uses the GUI to select the subset of an object state to be marshaled. Figure 2.2 displays class `Mass` from Section 3, with fields `mass`, `x`, `y`, and `surrounding` selected for marshaling. This visual input is the only action required from the programmer to parameterize the backend code generator of our tool, which then generates marshaling functionality. The programmer can then simply include the automatically generated marshaling code with the rest of their HPC application.

2.3.1 User Interface

The programmer starts by selecting a C++ source file using a standard file browse dialog. In response, the tool invokes a C++ parsing utility called GCCXML [40]. This utility taps into the platform's C++ compiler (e.g., GCC on UNIX or Visual C++ on Windows) to create a structured XML description of a given C++ compilation unit; this XML description can be used by other language processing tools. Our tool then parses the XML file and displays the extracted information to the user through a GUI.

The GUI employs a tree-view visualization [44] to display C++ object graphs. The tree-view provides an intuitive interface for the programmer to explore the structure of an object graph and to select the fields to be included into the subset of the marshaled object's state. The root node of the tree-view represents the main class selected for marshaling. All other nodes represent fields transitively reachable from the main class. To prevent circular references from causing infinite node expansion, the tree-view implements a lazy node visualization strategy, expanding a node only when it is selected for marshaling.

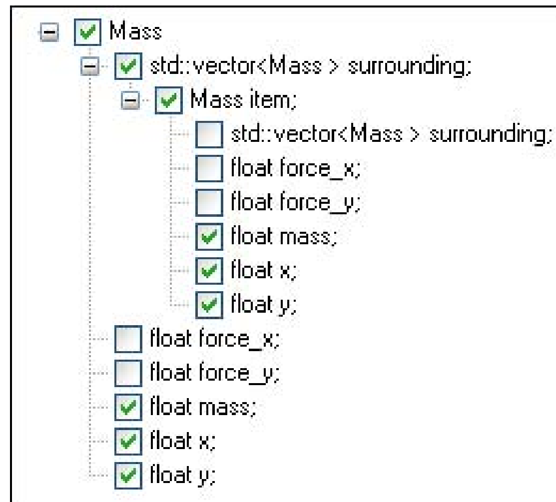

Figure 2.2: Selecting a subset of an object graph in MPI Serializer.

2.3.2 Handling C++ Language Features

Automatic marshaling of C++ data structures presents many challenges arising as a result of the sheer complexity of this mainstream programming language. In designing our tool, we had to address a number of these challenges to support a reasonable subset of the language. In the following discussion, we highlight the novel insights we have gained from designing our approach and automated tool.

Dynamic arrays

In C++, the meaning of a pointer variable is ambiguous. A pointer may be pointing to the memory location of a single element, or it may be pointing to the first element in a dynamic array. In the latter case, the language provides no standard way to determine the array's size. By convention, when designing a class, many C++ programmers define an additional member variable to keep track of the size of its corresponding dynamic array member variable. Our strategy for marshaling dynamic arrays assumes that the programmer is following this convention. To this end, the tool's GUI enables the programmer to visually

disambiguate the meaning of a pointer variable to be marshaled. If the variable is indeed a dynamic array, the programmer can select the numeric field representing the array's size.

Polymorphic fields

Another complication that pointer fields present for marshaling is polymorphism. A pointer to a base class may actually be assigned to an instance of any of its subclasses. Our tool automatically determines whether the possibility for polymorphism exists by examining the inheritance graph of each field. In lieu of a precise static analysis, the programmer is expected to specify object graph subsets for all possible derived instances to which a pointer field could be pointing at runtime. Thus, the programmer implicitly disambiguates the range of polymorphic possibilities for a pointer field. In the case of multiple possibilities, runtime type information (RTTI) is employed in the generated code to determine the appropriate marshaling functionality for the polymorphic field.

Non-public and Inherited fields

C++ supports the encapsulation principle of object-oriented programming by disallowing outside entities (i.e., other classes and methods) from accessing non-public fields of a class. However, non-public fields may be part of the subset of an object's state selected for marshaling. Previous directly related approaches employed two different strategies for accessing non-public fields for marshaling.

The Boost libraries require declaring their library `Archive` class as a `friend` to all marshaled classes. In C++, a `friend` entity is granted access to the non-public fields of a befriended class. This strategy is not acceptable for our approach, as it requires modification to the existing source, thereby violating our design objectives. C++2MPI uses a different strategy

by creating an isomorphic copy of all marshaled classes, replacing all non-public declarations with public ones in the replicated classes. The non-public fields' offsets in the original classes can subsequently be obtained by consulting the corresponding offsets in the replica classes at runtime. While this strategy does not require changes to the existing source code, the C++ standard [56] only requires that non-bit-fields of a class or structure without an intervening access-specifier (i.e., `private`, `protected`, or `public`) be laid out in non-decreasing address order. In other words, a C++ compiler is free to allocate blocks of fields with different access specifiers in an arbitrary order. As such, the C++2MPI strategy is not guaranteed to work for all C++ standard-compliant compilers. This strategy also results in code bloat: every marshaled class has to be deployed with its isomorphic copy. Such code bloat could be detrimental for the locality of reference in the processor's cache, resulting in performance overhead, which is unacceptable for our approach. Thus, solving the challenge of accessing non-public fields requires a new strategy capable of maintaining the existing source code without sacrificing performance.

Our approach to calculating the offsets of all fields of a C++ class ensures cross-platform and compiler independence while still strictly adhering to the C++ standard. This design is enabled by adding an additional code generation layer on top of the existing infrastructure of our tool. This layer generates metadata about the marshaled C++ data structures using a technique that combines the strategies of C++2MPI and the Boost libraries.

For each class, our approach generates an isomorphic copy that preserves the order of fields and access specifiers but also includes a `friend` declaration for a generated "field accessor" class. The field accessor class creates a list of all fields' offsets in a class using the standard `offsetof` macro. In C++, `friend` declarations are strictly compile-time concepts and have no affect on the memory layout of a class or its instances. Therefore, the field offsets obtained from a generated isomorphic class are guaranteed to be the same as the corresponding field

Original Class

```
class Mass {

private:
vector<Mass> surrounding ;

protected:
float force_x, force_y ;

public:
float mass; //mass value
float x, y ; //position
. . .
} ;
```

Generated Isomorphic Class

```
class Mass {
friend class OffsetGenerator;
private:
vector<Mass> surrounding ;

protected:
float force_x, force_y ;

public:
float mass; //mass value
float x, y ; //position
. . .
} ;
```

Field Accessor Class

```
class OffsetGenerator {
 . . .
 void createOffsetsList ( ) {
   int offset = offsetof(Mass, force_x);
   writeOffset("Mass.force_x", offset) ;
   . . .
 }
}
```

Offsets List

```
Mass.force_x, 16
...
```

Generated Marshaling Code

```
MPI_Datatype workerToMaster;

void createDatatypes (Mass* mass, . . .) {
   MPI_Aint addresses[2] = {. . .};
   . . .
   //add force_x to this datatype
   char* force_x = ((char*)mass) + 16;
   MPI_Address((float*)force_x, addresses[0]);
   . . .
   //register the datatype
   MPI_Type_struct(. . ., &workerToMaster);
   MPI_Type_commit(&workerToMaster);
}
```

Figure 2.3: Accessing non-public fields using our C++ standard-compliant strategy.

offsets in the original class.[1]

The obtained offsets are then inserted into the generated marshaling code to access an object's non-public fields. A slightly simplified example of accessing the `protected` field `force_x` in class `Mass` is shown in Figure 2.3. This approach minimizes runtime overhead by using generated constant offset values to access non-public fields, maintains a cache-friendly memory footprint by using isomorphic classes only at generation-time, and is guaranteed to be cross-platform and compiler independent by following a C++ standard-compliant strategy.

One could argue that our solution for accessing non-public and inherited fields violates the encapsulation principle by allowing an outside entity (i.e., the marshaling method) to access the non-public state of an object. However, the current solution is acceptable for the HPC domain, which has the requirements of preserving the existing source code and not incurring performance overheads take priority over strict adherence to object-oriented principles.

[1]Our approach relies on two assumptions. First, either the source or a corresponding GCCXML descriptor file must be provided for all third-party libraries. Note that the latter option is reasonable for closed-source libraries since the GCCXML file describes only the structure of the library and not any proprietary logic. Second, the C++ compiler must be deterministic. That is, the memory layout of two identically-named isomorphic classes will be the same. Although this property is not strictly guaranteed, it would likely be infeasible for a non-deterministic C++ compiler to be standards-compliant.

2.3.3 Generating Efficient Marshaling Code

In addition to the ability to support a large subset of the C++ language, our approach also implements a fast and memory-efficient buffer packing strategy. This strategy minimizes the required size of the buffer, reducing the bandwidth needed to transmit marshaled data. The key to the efficiency of our approach is generating code that leverages the mature marshaling facilities of MPI. Therefore, we first give a quick overview of these facilities before describing the novel insights of our strategy. For a detailed listing of the marshaling code generated to send a `Mass` object from Master to Worker and Worker to Master, see appendix A.

MPI provides a variety of API facilities to support marshaling. However, for this discussion, we focus on `MPI_Datatype` and `MPI_Pack`. The `MPI_Datatype` construct provides a reusable cached collection of static memory offsets for efficient marshaling of arbitrary data structures. The programmer first initializes an instance of a data structure and calculates the memory offsets and sizes of the fields to be marshaled. These offsets are then stored in an `MPI_Datatype` for later use.

To pack data into a buffer, MPI provides an API function `MPI_Pack`. The user passes the data structure to be marshaled and the buffer to store the marshaled data. However, `MPI_Pack` supports only primitive C++ types and user-defined `MPI_Datatype`'s. For objects containing dynamically allocated fields, using `MPI_Pack` is not trivial, as it requires static offsets for all the marshaled data.

To guide the generation of efficient marshaling code, our approach first performs a bounds-checking operation on the marshaled object graph. Bounds-checking traverses the object graph in order to determine if the user-specified subset of the object's state is "bounded." A bounded object state is a transitive property, signifying whether memory offsets and sizes of all marshaled fields can be determined statically. An object graph containing dynamic

array, pointer, or STL container fields violates this property.

Since using `MPI_Datatype` is known to be more efficient for packing bounded objects than packing fields individually [68], our approach utilizes this construct for marshaling all bounded subsets of an object graph. Figure 2.4 illustrates how our approach utilizes `MPI_Datatype`.

If the programmer selected an unbounded subset of an object graph, our approach then employs static polymorphism to provide a global method `MPI_Pack` with a similar signature to that of the standard `MPI_Pack` method. While the standard method takes an `MPI_Datatype` as a parameter, the generated polymorphic method takes an automatically-generated `MPI_Descriptor` enum type specifying the marshaling strategy to use.

This approach is particularly useful for non-experts, who can follow a uniform convention by calling both the standard `MPI_Pack` on a bounded object with an `MPI_Datatype` and by calling the generated `MPI_Pack` on an unbounded object using an `MPI_Descriptor`. The distinction between the standard and the generated version of `MPI_Pack` is resolved at compilation time, resulting in zero runtime performance overhead.

2.3.4 Using Generated Code

To further illustrate how the programmer uses generated marshaling code, consider the Master-Worker communication that needs to pass instances of class `Mass`. In the case of passing an instance of `Mass` from Worker to Master, the subset of the object graph consists of `force_x` and `force_y`, and as such is entirely bounded. To maximize efficiency, the tool generates an `MPI_Datatype` (`workerToMaster`) for this subset. However, passing an instance of `Mass` from Master to Worker, the marshaled subset is unbounded, as it contains an STL container field, `surrounding`. This subset, therefore, requires custom marshaling code, capable of traversing the unbounded subset efficiently. The generated code consists of

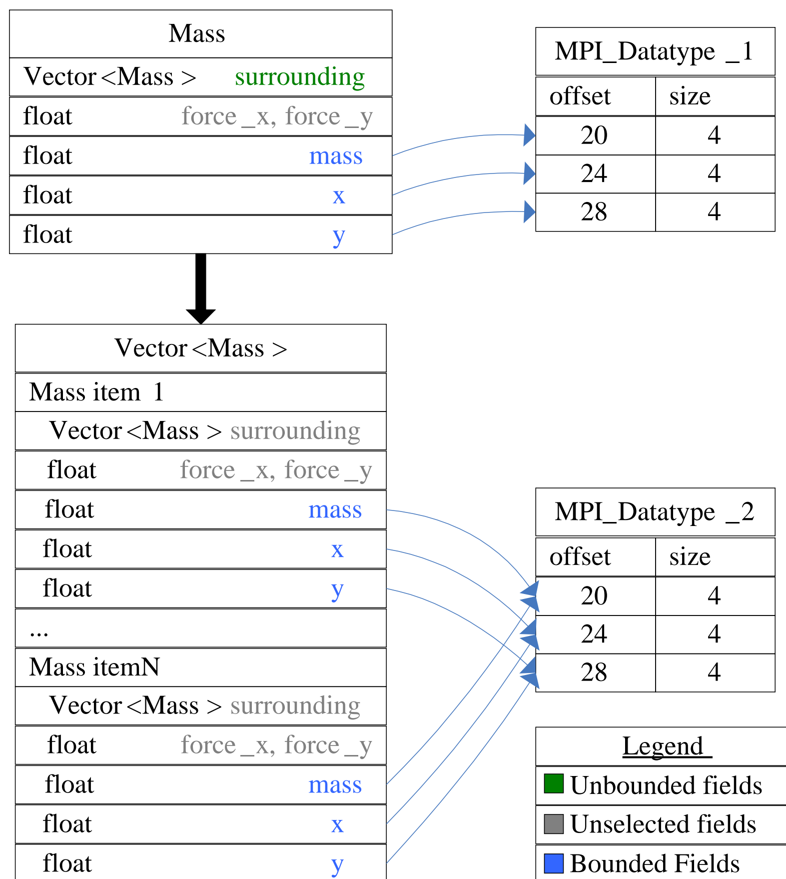

Figure 2.4: An example of mapping bounded subsets of an object graph to `MPI_Datatype`'s. The offsets and sizes are for a 32-bit architecture using GCC.

a custom `MPI_Pack` function and an `enum` type, `masterToWorker`. However, the marshaling interface exposed to the programmer is almost identical in bounded and unbounded cases, as shown below:

```
void foo() {
  Mass mass; char* buf;
   ...
  //pack for Master to Worker
  MPI_Pack(&mass,1,masterToWorker,buf,...);
   ...
  //pack for Worker to Master
  MPI_Pack(&mass,1,workerToMaster,buf,...);
}
```

The two separate packing calls are exposed to the programmer as if they were the same `MPI_Pack` call, even though the call for `masterToWorker` actually invokes a specialized generated packing method. The advantage of this approach is that it enables the programmer to use the generated code almost identically to how they use the standard MPI interface.

2.4 Case Studies and Performance Results

The purpose of the presented case studies and the associated performance comparisons is to validate our approach against the requirements stated in Section 2.1 as well as against the existing state of the art. To recap, our goal is to support a sufficiently large subset of the C++ language, while ensuring high performance in the generated marshaling code.

The evaluation of MPI Serializer that we have conducted consists of micro-benchmarks and case studies. The micro-benchmarks enable us to pinpoint the fine-grained performance

advantages and limitations of our approach. The two case studies involve the automatic generation of the marshaling functionality for real HPC applications. The first case study showcases the use of our tool as a refactoring aid in the often difficult task of parallelizing a sequential program. The second case study demonstrates the fitness of our tool to provide efficient marshaling functionality for an existing high-performance application.

In making the performance related arguments, we compare our work against the Boost libraries and hand-written code on a 3.0GHz dual-core Pentium 4 with 2GB of RAM running Debian GNU/Linux, GCC version 4.1.2. However, a comparison of our work with other directly related approaches, such as MPI Pre-Processor [77], AutoMap/AutoLink [42], or C++2MPI [46], is impossible due to their lack of support for C++ language features or even C++ itself.

One additional advantage of our approach that is easy to overlook by focusing on performance numbers is that it reduces the amount of maintained hand-written source code. Since the complexity of software grows exponentially in relation to the size of a program [9], every line of source code that the programmer has to write by hand contributes to the software maintenance burden. Addressing changed requirements or fixing program defects requires a program maintenance effort that is directly proportional to the size of a program [43]. By reducing the amount of maintained source code, our approach has the potential to ease the software maintenance burden. Therefore, while the following case studies highlight performance gains, our approach also provides software engineering benefits to the target applications.

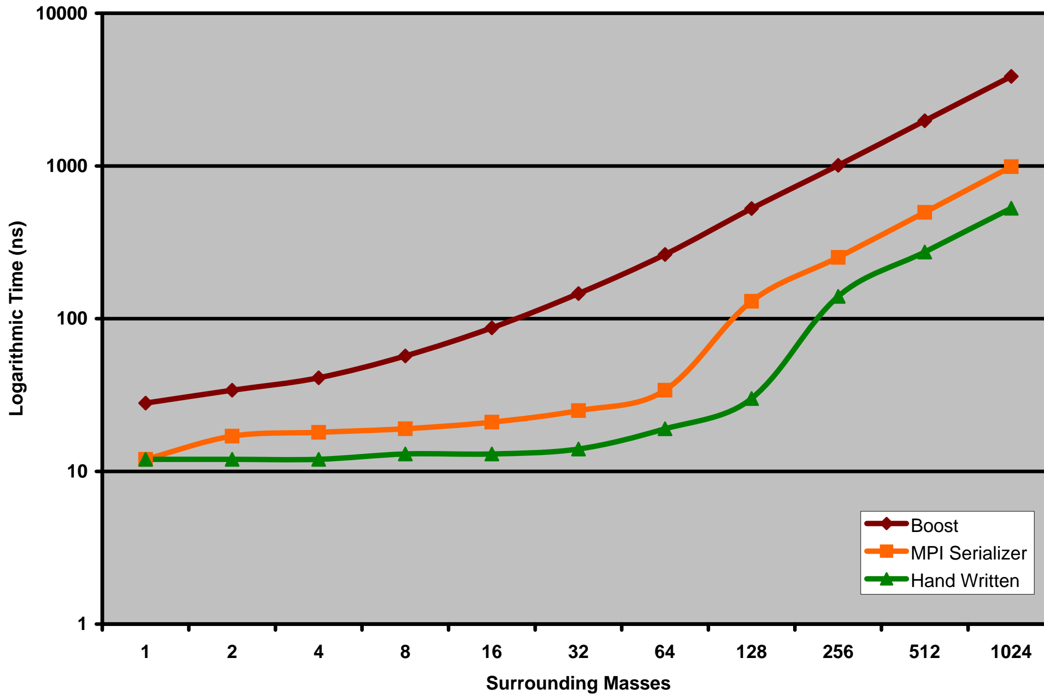

Figure 2.5: Benchmark results comparing the total marshaling and unmarshaling time required for the `Mass` class example.

2.4.1 Micro-benchmarks

For the micro benchmark, we used the `Mass` class described in Section 3. Figure 2.5 shows the total combined marshaling, sending, and unmarshaling time taken for the Master to Worker communication as described earlier. The x-axis represents the number of surrounding `Mass` objects (i.e., being marshaled and unmarshaled). The figure demonstrates the differences in performance between hand-written code utilizing the Boost libraries and code automatically generated by our tool. The results show a speed-up between 2x and 4x for our approach compared to Boost, with the rate of speedup increasing as the size of the transmitted data structure grows.

While the generated code is highly efficient, the programmer can still write fine-tuned mar-

shaling code by hand, which would yield better performance. This benchmark explores an optimization technique for marshaling collections called striding. It refers to using a heuristic to reduce the number of collection elements to be marshaled. The results show that using a basic striding increment of 3 (i.e., selecting only 1/3 of the elements in the vector of `Mass` objects) can still beat our approach.

2.4.2 Parallelizing NEAT

NeuroEvolution through Augmenting Topologies [85] (NEAT) is an artificial intelligence algorithm for training a collection of artificial neural networks. NEAT is a genetic algorithm [5] that mimics Darwinian evolution by repeatedly competing potential solutions against each other and then selecting and breeding the fittest individual solutions. We used MPI Serializer to automatically generate marshaling code for an on-going research project that parallelizes NEAT to run on a supercomputer [101].

Figure 2.6 shows the time required to marshal and umarshal a NEAT population set using the Boost libraries compared to using our approach. The x-axis in Figure 2.6 represents the number of potential solutions (i.e., population elements) to be marshaled. The performance numbers indicate similar scalability for both approaches. However, the abstractions provided by the Boost libraries result in a performance overhead causing our approach to be as much as an order of magnitude faster in some cases. Additionally, this particular application requires the marshaling and unmarshaling of several nested fields (i.e., object fields with other object fields). Providing Boost `serialize` methods by hand is much more difficult for this case, as several classes must be modified. By contrast, our approach requires the programmer to manipulate only a single visual hierarchy.

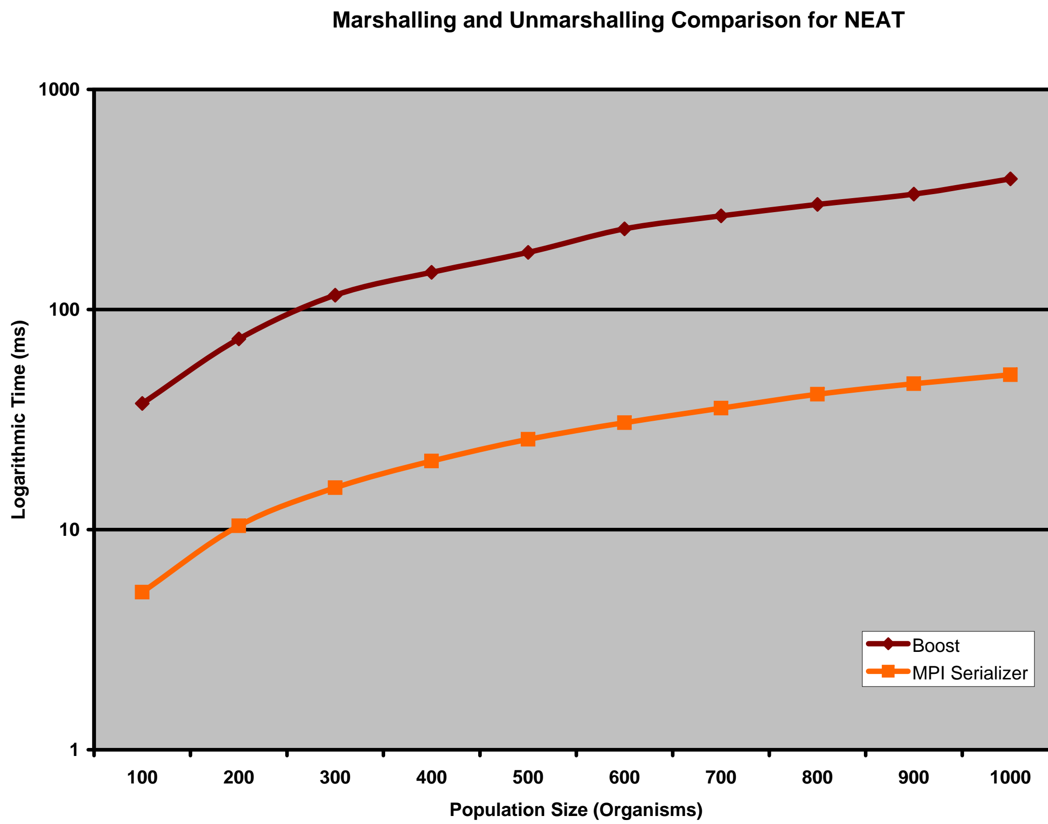

Figure 2.6: Benchmark results comparing the marshaling time required for a population of organisms in NEAT.

2.4.3 mpiBLAST 2.0

mpiBLAST [29] is a widely-used, open-source, parallel application for aligning genome sequences. It solves the problem of finding the closest known matching sequence for a given genome. mpiBLAST has been known to assist in the process of scientific discovery in domains as diverse as new drug development and classifying new virus species. mpiBLAST 2.0 is written in C++ [3], with MPI as the communication middleware. In this case study, we re-implemented the hand-written marshaling functionality of mpiBLAST by using both the Boost libraries and our automated tool. For benchmarking, we chose the phase of the mpiBLAST algorithm when the Worker processes report back their search results to the Master.

Figure 2.7 compares the performance between the original hand-written code, the code using the Boost libraries, and the code automatically generated by our tool. The x-axis represents the number of sequence result collections marshaled. The results show that the automatically generated code is slightly more efficient than hand-written code and nearly an order of magnitude faster than the Boost implementation in some cases.

Similar to the NEAT case study, Boost's abstractions once again account for the overhead incurred while marshaling and unmarshaling. More surprisingly, the generated code yielded better performance than the original hand-written implementation. The cause of this is that the original implementation currently uses a less-efficient stream-based strategy to marshaling and umarshaling, to aide in the debugging process. However, the code generated automatically by our tool should require no debugging (provided that the tool's implementation is mature enough), as long as the visual input specified by the programmer correctly reflects the subset of the object graph to be marshaled.

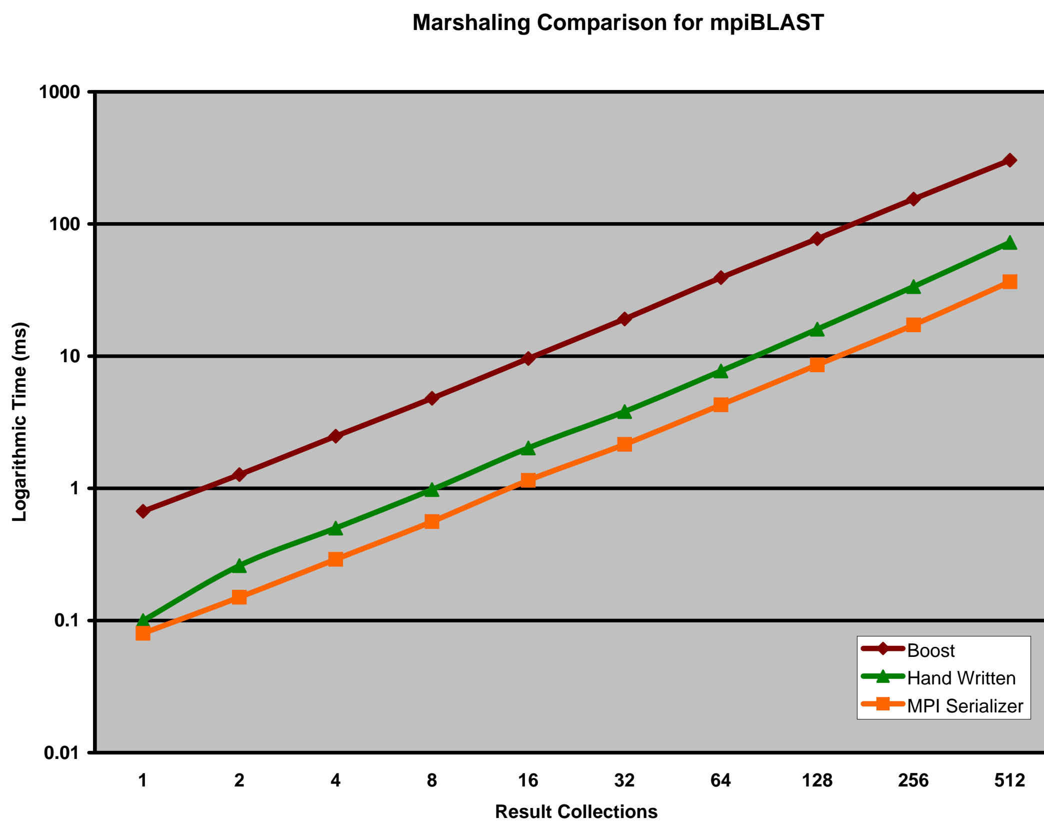

Figure 2.7: Benchmark results comparing the marshaling time required for a collection of genome alignment results in mpiBLAST.

2.5 Future Work

The target audience of our approach and automated tool is non-expert HPC programmers. Therefore, to further increase the usability of our tool, we would like to focus on improving the software engineering quality of the generated code. First, by replacing `MPI_Pack` calls with size calculations, the same approach can be used to automatically determine the exact size of a destination buffer. Integrating this technique will make it possible for the generated code to provide a more programmer-friendly external interface, similar to that of Boost.

Our approach is not necessarily divergent from Boost and other marshaling libraries. The clean, object-oriented interface that the Boost libraries provide to MPI programmers may in some cases be worth the accompanying overhead. Consequently, as a future extension, we plan to provide the capability to generate Boost `serialize` methods for data structures. This demonstrates that our approach's intuitive visual interface does not need to generate marshaling code which is difficult to read or modify by hand if necessary.

Finally, although our tool generates code specifically for HPC applications using MPI, our generalized approach extends beyond this specialized domain. Examples of other applicable software development scenarios that could benefit from having automatically generated marshaling code packed to a generic buffer include transmission, permanent storage, and check-pointing.

2.6 Conclusions

Our approach provides automatic synthesis of efficient marshaling functionality for HPC applications based on visual input supplied by the programmer. This aspect of our approach is particularly appealing to the many lab scientists and engineers who have limited parallel

programming experience. By automatically generating low-level, high-performance marshaling code, our approach eliminates the need to write tedious and error-prone code by hand, thereby facilitating the process of scientific discovery.

Chapter 3

Rosemari

3.1 Introduction

Despite the benefits of annotation-based frameworks, several major drawbacks hinder their adoption and use. Legacy applications that were developed using older framework versions based on type and naming requirements must be upgraded to the annotation-based versions. This upgrade often requires hundreds or even thousands of tedious changes to source files scattered throughout the code base. While these changes are intuitively obvious to the programmer, automating them is not trivial. A text-based find-and-replace approach is not sufficient, as the required changes cannot be correctly detected with a regular expression search. Furthermore, existing inference algorithms cannot detect them automatically, and writing an automated refactoring tool by hand can be time-consuming and error-prone. These complications often force a manual refactoring in order to upgrade an application.

This chapter presents a novel refactoring approach that solves both the Version and Vendor Lock-In problems commonly associated with annotation-based frameworks. Our approach

has three phases: first, the framework developer creates representative examples of a class before and after transitioning; second, our algorithm infers generalized transformation rules from the given examples; finally, application developers can use the inferred rules to parameterize our program transformation engine and automatically refactor their legacy applications.

We validate our approach by inferring refactorings for transitioning between three different unit testing frameworks (JUnit 3 [8], JUnit 4, and TestNG [10]), as well as three different persistence frameworks (Java Serialization, Java Data Objects (JDO) [82], and Java Persistence API (JPA) [30]). With only five minor refinements to the inferred unit testing transformation rules, we automatically upgraded more than 80K lines of testing code in JHotDraw, JFreeChart, JBoss Drools, and Apache Ant from JUnit 3 to JUnit 4.

This chapter makes the following novel contributions:

- *Annotation Refactoring*– a new class of refactorings that replaces the type and naming requirements of an old framework version or annotation requirements of a different framework with the annotation requirements of a target framework.
- An approach to removing the Version and Vendor Lock-in anti-patterns for annotation-based frameworks.
- A differencing algorithm that accurately infers general transformation rules from two versions of a single example.

3.2 Motivating Example

Whenever a widely-used framework undergoes a major version upgrade (i.e., changing the structural requirements for application classes), framework developers or other domain ex-

perts commonly release an upgrade guide. A typical presentation strategy followed by such guides is to show an example legacy class and its corresponding upgraded version. With respect to annotation refactorings, recent framework upgrades that have led to the creation of such guides include Enterprise JavaBeans (EJB) version 2 to 3 [72], Hibernate annotations to the Java Persistence API (JPA) [100], and JUnit version 3 to 4 [86].

JUnit 3

```
import junit.framework.*;

public class ATest extends TestCase
{

  //called before every test
  protected void setUp() {}


  //called after every test
  protected void tearDown() {}


  //tests Foo
  public void testFoo() {}


  //tests Bar
  public void testBar() {}
}
```

JUnit 4

```
import org.junit.*;

public class Atest
{
  //called before every test
  @Before
  public void setUp() {}

  //called after every test
  @After
  public void tearDown() {}

  //tests Foo
  @Test
  public void testFoo() {}

  //tests Bar
  @Test
  public void testBar() {}
}
```

Figure 3.1: Comparisons of a JUnit test case in version 3 and 4.

To illustrate the challenges associated with annotation refactorings, consider upgrading a JUnit application from version 3 to 4. Figures 3.1 and 3.2 show two example legacy classes and their corresponding upgraded versions, derived from the upgrade guide [86]. Application classes that use JUnit fall into two categories: test cases and test suites. A test case class contains a set of methods for testing a piece of application functionality. A test suite

JUnit 3	JUnit 4

`//gets a collection of two`
`//test cases to run`
`public class AllTests`
`{`
`  public static Test suite() {`
`    TestSuite suite =`
`      new TestSuite(ATest.class);`
`    suite.addTestSuite(BTest.class);`
`    return suite;`
`  }`
`}`
	`@RunWith(Suite.class)`
	`@SuiteClasses({ATest.class,`
	`               BTest.class})`
	`public class AllTests`
	`{`
	`  //no suite method required`
	`}`

Figure 3.2: Comparison of a JUnit test suite in version 3 and 4.

class groups related test cases, so that the framework can invoke them as a single unit. Applications using JUnit 3 must adhere to type and naming requirements to designate test cases and suites. However, JUnit 4 switched to using an annotated POJO paradigm to express the same functionality. Table 3.1 summarizes the differences between the application requirements of the two versions.

Application Element	JUnit 3	JUnit 4
Test case class	Extend `TestCase`	None (POJO)
Initialization method	Override `setUp`	`@Before`
Destruction method	Override `tearDown`	`@After`
Test method	Name starts with “test”	`@Test`
Test suite class	Provides `suite` method	`@RunWith`, `@SuiteClasses`

Table 3.1: A summary of the requirements for application elements using JUnit versions 3 and 4.

While the refactorings required to upgrade a JUnit application may be straightforward to the programmer, no existing refactoring tool can make these changes automatically. Therefore, if the framework developer wishes to provide automated upgrade support, she often has no choice but to create a refactoring tool by hand. Such a task may require a significant

investment by the framework developer, as she must first gain proficiency in a refactoring library API (e.g., the Java Development Toolkit (JDT) [36]) or a domain-specific language (e.g., Spoon [73]). After becoming familiar with a library or a DSL, she must then use it to write the refactoring tool from scratch. As a result, framework developers typically opt to provide backwards compatibility support rather than automated upgrade tools. Although providing backwards compatibility allows legacy applications to use a newer version of a framework, it does not allow them to take advantage of newly-introduced framework features.

Thus, an approach to automatically generating refactoring tools capable of upgrading legacy applications has great potential benefit. Next we provide an overview of our approach and show how it can be used to automatically generate a refactoring tool for upgrading JUnit applications.

3.3 Approach Overview

Our approach is based on the wide availability of upgrade guides containing examples of legacy classes and their upgraded versions. If the human developer is expected to infer general rules for upgrading their legacy applications using these guides, then the examples are likely to contain a level of detail that one could leverage to automate the process. Our approach automatically extracts this knowledge, creating specialized refactoring tools that can upgrade legacy applications. Our Eclipse Plug-in called Rosemari (**R**ule-**O**riented **S**oftware **E**nhancement and **M**aintenance through **A**utomated **R**efactoring and **I**nferencing) implements our approach.

Creating an annotation refactoring tool involves three steps. First, the framework developer selects two versions of a class, one before and one after upgrading, that we call *representative examples*. The developer then picks a predefined specialization of our inference algorithm,

called an *upgrade pattern*. Finally, the generated rules can be downloaded by application developers and subsequently used to parameterize our transformation engine, which will then automatically refactor their legacy applications. Next we detail the main steps of our approach, using the previously mentioned JUnit example for demonstration.

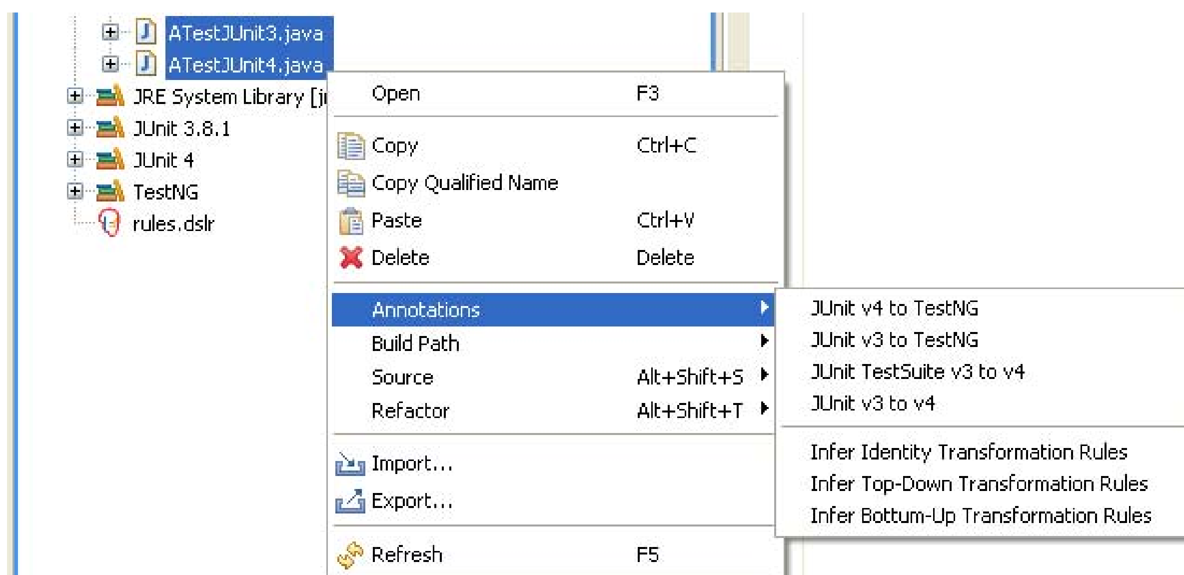

Figure 3.3: The Rosemari context menu. The top set of menu items are available refactorings; the bottom set of menu items are available inference patterns.

3.3.1 Representative Examples

A representative example is a class or interface, such as those commonly included in framework tutorials, that uses framework features that differ between versions or vendors (e.g., Figures 3.1 and 3.2 can serve as representative examples for upgrading JUnit). A representative example using an older framework is called a *prior example*. A representative example using a newer framework is called a *posterior example*. The framework developer provides a prior example and its corresponding posterior example, which may differ in the following ways:

1. Super type changes
2. Method signature changes
3. Field type changes
4. Annotations added or removed
5. Annotation argument added or removed
6. Imports added or removed
7. Statement added or removed

The prior and posterior examples are compared at different levels of granularity in the encapsulation hierarchy, which we call *levels* for short. Specifically, examples are compared at the class, method, and statement levels. Differences between levels are used to detect the restructurings required to upgrade the prior example into the posterior example. However, detecting restructurings is only one facet of creating a refactoring. The decision of when to apply a restructuring to a level requires additional knowledge about how the application is being upgraded.

```
public class D extends C
{

      public void foo() {
            System.out.println();
      }
}
```

```
public class D extends C
{
      @Ann
      public void foo() {
            System.out.println();
      }
}
```

Figure 3.4: An example showing the ambiguity inherent in inferring refactorings between two arbitrary representative examples.

For instance, consider the simple pair of prior and posterior examples in Figure 3.4. While it is obvious that there is a method-level difference that requires adding the `@Ann` annotation

to `foo`, the refactoring rule to be inferred is unclear. One possible rule may be that a method named "foo" in any subclass of `C` should be annotated with `@Ann`. However, another rule could be that any method which calls `System.out.println` should be annotated with `@Ann`. To disambiguate when a restructuring should be applied in a refactoring, we follow a pattern-based approach described next.

3.3.2 Upgrade Patterns

When upgrading from a type and naming convention-based framework version to a newer annotation-based version, or when switching between different annotation-based frameworks, refactoring rules typically follow common patterns, which we call *upgrade patterns*. One reason why these patterns occur is that evolving a framework to use annotations is normally driven by the desire to improve the software engineering quality of the framework (e.g., looser coupling between application and framework classes). Another reason is that switching between different annotation-based frameworks often requires using a different vocabulary to describe essentially the same functionality.

Therefore, inferring the refactorings between two representative examples depends on the upgrade pattern followed. Next we describe three such common patterns we have identified from our experiences. While we have found that these patterns successfully capture the refactorings for many upgrade scenarios, other patterns can be plugged into our system as needed.

Bottom-Up

This upgrade pattern applies restructurings on the basis of the level itself and its enclosing levels. For example, refactoring the methods in a JUnit 3 test case follows a bottom-up

pattern. Specifically, the `test`, `setUp` and `tearDown` method restructurings are only relevant if their enclosing class is a `TestCase`.

Top-Down

In contrast to the bottom-up pattern, a top-down pattern applies restructurings on the basis of the level itself and its contained levels. For example, refactoring test suite classes in JUnit 3 follows a top-down pattern. Specifically, the class is annotated with the `@RunWith(Suite.class)` annotation on the basis of containing a `suite` method.

Identity

This upgrade pattern applies restructurings only on the basis of the level itself, assuming that all annotations in the first representative example have a one-to-one mapping in the second representative example. For instance, the TestNG framework uses `@BeforeMethod`, which is identical in its functionality to the `@Before` annotation in JUnit 4.

The developer can choose an appropriate upgrade pattern by observing the purpose of a given annotation. If adding an annotation removes tight coupling between the *enclosing* elements of a level (e.g., a class enclosing a method) and the framework, then it is likely a bottom-up pattern. If, however, the annotation describes the *contained* elements of a level (e.g., a method contained in a class), then it is likely a top-down pattern. If the annotation simply changes how the level is expressed, then it is likely an identity pattern.

3.3.3 Transformation Rules

Inferred refactorings are represented as a collection of first-order *when-then* transformation rules,[1] expressed using a Domain Specific Language (DSL) we developed. A transformation rule is composed of restructurings and constraints on their application. The *when* portion of a rule defines the constraints under which to apply the restructurings defined in the *then* portion of the rule.

While the generated rules possess a high degree of precision, some rules may require manual refinement. Since the developer is not expected to write the transformation rules from scratch but rather fine-tune the generated rules, we chose to trade conciseness for readability and ease of understanding in designing our DSL. Having a collection of generic natural-language statements that can be parameterized with concrete values, the developer can easily refine complex rules to better express the desired refactorings.

Given the JUnit test case representative examples in Figure 3.1 and the Bottom-Up upgrade pattern, Rosemari generates five transformation rules, shown in Figure 3.5. Rules 2-4 originally require that the target class directly extends `TestCase`. However, these transformations are valid for any class that extends `TestCase`, directly or indirectly. Therefore, the developer can change this constraint in each of the three rules by refining it to its more general version: *Superclass is a "junit.framework.TestCase"*. Additionally, the *Visibility is protected* constraint in rules 2 and 3 can easily be generalized by changing the constraint to *Visibility is at least protected*.

The Rosemari plug-in provides a searchable collection of supported natural language statements that the developer can reference when refining transformation rules. Once refined and named appropriately (e.g., *JUnit v3 to v4*), the refactoring can be applied by selecting a

[1] We use the JBoss Drools engine[76].

JUnit 3 test case source file and choosing the refactoring from the context menu, as shown in Figure 3.3. The source file will then be automatically upgraded to JUnit 4.

Next we present our algorithm for inferring annotation refactoring rules.

3.4 Inference Algorithm

Our algorithm accepts two representative examples and infers a generalized set of transformation rules that compose a refactoring. We present our algorithm as a collection of seven smaller set manipulation algorithms configurable through a user-defined partial order on the levels of a representative example. To further illustrate how our algorithm works, we show how each component algorithm contributes to learning the transformation rules for a simple Enterprise JavaBeans (EJB) upgrade scenario. Figure 3.6 shows the two EJB representative examples, derived from the Oracle guide [72] on migrating applications from EJB 2 to 3.[2]

3.4.1 Decomposing Representative Examples

In order for our algorithm to calculate transformation rules, each representative example must first be decomposed into a set of levels, $P = \{L_1, L_2, \ldots, L_n\}$. A *level* is a set, $L = \{e_1, e_2, \ldots, e_m\}$, of signature elements (i.e., tokens in a program element's signature) at a particular point in the encapsulation hierarchy. For example, the method level contains a set of annotations, a visibility identifier, a scope identifier, a return type, a set of parameters, and a set of exceptions. Figure 3.7 shows a full list of the supported levels and their definitions.

As discussed in Section 3.3.2, inferring the correct transformation rules given a pair of rep-

[2] Fully upgrading EJB applications requires static analysis of both Java source code and XML metadata files and is thus outside the scope of this work. The rules learned in this example are therefore only a subset of the total required to refactor an entire EJB application.

resentative examples is not possible without additional information about the refactoring. The Rosemari plug-in implementing our algorithm uses the concept of upgrade patterns to simplify this process. However, upgrade patterns are merely wrappers for specifying a partial order over the set of levels in a representative example. Figure 3.8 shows the partial orders corresponding to each of the three previously discussed upgrade patterns. Figure 3.9 shows the level sets for the EJB example, which matches to a Bottom-Up pattern.

3.4.2 Component Inference Algorithms

Our algorithm is to constructs sets of independent components and merges them hierarchically. This section presents the five sub-algorithms that compose these independent components.

LevelRestructurings

Our algorithm starts by first calculating the restructurings for each level. A *restructuring* is a simple program transformation function that adds, replaces, or removes a signature element in a level. We call a restructuring a *positive restructuring* if it adds or replaces an element. A restructuring is a *negative restructuring* if it removes an element. Figure 3.10 presents the LevelRestructurings algorithm for discovering a level's restructurings.

The Prune method (line 6) removes unnecessary elements from $NEGELS$. This is an implementation addition to reduce the number of negative restructurings generated, making the final transformation rules more concise and readable. For instance, changing the visibility of a level is done by replacing the current visibility with the new visibility (a positive restructuring) rather than first removing the current visibility (a negative restructuring) and then adding the new visibility (a positive restructuring).

For the EJB example, LevelRestructurings is called three times (i.e., once for each level). For the *HelloWorldBean* level, $POSELS_{HelloWorldBean} = \{A^{Stateful}\}$, and thus a positive restructuring of *Add annotation "Stateful"* is generated. Similar positive restructurings are generated for the *ejbActivate* and *ejbRemove* levels. However, only the *HelloWorldBean* level produces a non-null value for removed elements, as $NEGELS_{HelloWorldBean} = \{I^{SessionBean}\}$, resulting in a negative restructuring of *Remove interface "SessionBean."*

Lemma 1. Let $L = \{e_1, e_2, ..., e_m\}$ be a prior level, and let $L' = \{e_1, e_2, ..., e_n\}$ be a posterior level, such that L' is the restructured version of L. $LevelRestructurings(L, L')$ returns a set of restructurings, $R = \{r_1, r_2, ..., r_k\}$ representing exactly every required positive and negative restructuring to transform L to L'.

Proof. If there is an element, $e_u \in L'$ such that $e_u \notin L$, then a positive restructuring $r_i^+ = PositiveRestructuring\ (e_u)$ is required to transform L to L'. Thus, $POSELS = L' - L$ (line 2) will contain e_u, and adding a positive restructuring to R for every element in $POSELS$ (lines 3-4) guarantees that r_i^+ will be in R. If a positive restructuring, $r_j^+ = PositiveRestructuring(e_v)$ is not required to transform L to L', then either $e_v \in L$ or $e_v \notin L'$. If $e_v \in L$, then $L' - L$ will remove e_v and it will not be added to $POSELS$; likewise, if $e_v \notin L'$ then $L' - L$ and consequently $POSELS$ will not contain it. If $POSELS$ does not contain e_v, then r_j^+ will not be created in line 4 and $r_j^+ \notin R$. Therefore, R contains every positive restructuring and no more. An analogous and opposite argument applies to negative restructurings.

LevelConstraints

Our algorithm next computes a set of constraints defining when the discovered restructurings should be applied to a level. We call a constraint a *positive constraint* if it requires that an element be present. A constraint is a *negative constraint* if it requires that an element not be present. Negative constraints are necessary to ensure the generated transformation rules are not applied unnecessarily (e.g., to levels that have already been transformed by hand or a previous upgrading session). Figure 3.11 presents the LevelConstraints algorithm for discovering a level's constraints.

For the *HelloWorldBean* level, lines 2-3 create two positive constraints, *Visibility is public* and *Directly implements SessionBean*; lines 5-6 then add one negative constraint, *Not annotated with "Stateful"*. For both method levels, positive constraints requiring public visibility, member scope (i.e., non-static scope), and a return type of void are added along with negative constraints requiring *ejbActivate* and *ejbRemove* to not be annotated with *"PostActivate"* and *"Remove"*, respectively. Our implementation also handles special cases such as when a method does not contain any parameters, as with both EJB method levels which receive a negative constraint requiring that there are no parameters in the method signature.

Lemma 2. Let $L = \{e_1, e_2, ..., e_m\}$ be a prior level, and let $L' = \{e_1, e_2, ..., e_n\}$ be a posterior level, such that L' is the restructured version of L. $LevelConstraints(L, L')$ returns a set of constraints, $S = \{s_1, s_2, ..., s_k\}$ representing exactly every required positive and negative constraint to match L.

Proof. If there is an element, $e_u \in L$, it must have an associated positive constraint, $s_i^+ = PositiveConstraint(e_u) \in S$. All elements in L are iterated over and their corresponding positive constraints are added to S in lines 2-3. If a positive constraint $s_j^+ =$

$PositiveConstraint(e_v)$ is not required to match L, then $e_v \notin L$ and s_j^+ is never added to S. Thus, S contains exactly every positive constraint. If and only if there is an element $e_w \in L'$ such that $e_w \notin L$, it must have an associated negative constraint, $s_k^+ = PositiveConstraint(e_w) \in S$, since according to lemma 1 it will generate a positive restructuring in $LevelRestructurings$. $NEGELS$ will therefore contain e_w (line 4) and add a negative constraint, s_k^+, for exactly all elements satisfying the aforementioned constraint (lines 5-6).

NamingConvention

For method and field levels, our algorithm adds a name matching constraint intended to capture naming conventions. Figure 3.12 shows the *NamingConvention* algorithm which takes a set, $N = \{n_1, n_2, \ldots, n_m\}$, of fields or methods with identical signatures in both the prior and posterior examples and returns a generalized regular expression matching all the names in N. The algorithm first tokenizes a name based on the Java naming convention of uppercase delimiters, then calculates the tokens which match identically. If no naming convention is found, a literal expression matching only N is returned. As this is the case for both EJB method levels, each has an exact naming constraint added.

ContextConstraints

Although a level is context-free, program transformations are often not context-free operations but are rather based on some framework-dependent notion of context. To accommodate these scenarios, we introduce the notion of *context constraints*. A context constraint, q for level L_1 is a level constraint, $s \in L_2$, such that $L_1 <_L L_2$ where $<_L$ is a user-defined partial order on the set of levels, P. Figure 3.13 shows the ContextConstraints algorithm for discov-

ering all context constraints for a set of prior levels. It should be noted that the algorithm presented is only a semantically-equivalent version of the implementation, as in practice caching data structures can be used to eliminate the internal loop (lines 4-6).

Since our EJB example follows a Bottom-Up pattern, all method levels are defined as dependent on their enclosing class level. Thus, ContextConstraints adds the additional requirements that annotating a method named *ejbActivate* or *ejbRemove* with `@PostActivate` or `@Remove`, respectively, is only correct if the enclosing class has public visibility and implements `SessionBean`.

Lemma 3. Let $P = \{L_1, L_2, ..., L_n\}$ be a set of prior levels, let $<_L$ be a partial order on P, and let $S_{all} = \{S_1, S_2, ..., S_n\}$ be a set of sets of constraints, such that $\forall S_i \in S_{all}, \forall L_i \in P$, S_i is the set of context-free constraints for L_i. $ContextConstraints(P, S_{all}, <_L)$ returns an ordered set of sets of context constraints, $Q_{all} = \{Q_1, Q_2, ..., Q_n\}$ such that $\forall Q_i \in Q_{all}$, Q_i is exactly the set of contexts constraints for L_i.

Proof. It is necessary to first show that $\forall Q_i \in Q_{all}$, Q_i is a set of context constraints for L_i, then we show that Q_i is the *exact* set of context constraints as specified by $<_L$. *ContextConstraints* creates a new set of context constraints for every level in P (line 3) and adds the set to Q_{all} (line 7); thus, $|Q_{all}| = |P|$, and this completes the first half of the proof. For every Q_i, *ContextConstraints* iterates over all elements in P and adds a set of level constraints to Q_i if and only if their corresponding level satisfies $<_L$; thus, Q_i is exactly the set of context constraints for L_i, and this completes the second half of the proof.

SalienceValues

Context-dependent transformations hinge on the assumption that the context of the level they are transforming will not be invalidated before the level has been transformed. This assumption could not be guaranteed if the rules were executed arbitrarily, except in special circumstances [55]. For instance, the method-level transformations for the EJB example require that the target method is defined in a class implementing `SessionBean`. If the class-level transformations are arbitrarily executed before the method-level transformations, the interface will be removed and the method-level transformations will fail. To overcome this limitation, our algorithm calculates an execution order for each level, called a salience value. Rules with higher salience values will be executed prior to rules with lower salience values (ties are broken arbitrarily). Figure 3.14 shows the $SalienceValues$ algorithm for calculating salience values for a set of prior levels. The algorithm is analogous to a sorting algorithm parameterized by the partial order on L.

3.4.3 Merging Algorithms

Once the necessary independent components have been inferred, two merging algorithms are invoked. The first merging algorithm, $LevelTransformations$, combines the independent components for a level into a set of transformations for that level. The second merging algorithm, $ProgramTransformations$, combines the level transformations into a complete set of program transformations.

LevelTransformations

The $LevelTransformations$ algorithm takes the results of the algorithms described in the previous section for a single level, and combines them into a set of generalized transformation

rules. Figure 3.15 shows the $LevelTransformations$ algorithm. Until now, we have focused on structural inference at or above the level of method signatures. We have deliberately not detailed how we infer annotation attribute values, which can be any valid Java expression and as such require a deeper level of inference than that of method declarations. Thus, inference of attribute values is delayed until the $LevelTransformations$ algorithm (lines 4-10), when more complete program information is known. For every argument to the attribute, $LevelTransformations$ checks if a matching expression exists in the prior representative example (Lines 6-7). If it does not, the algorithm generates a literal transformation which adds the expression to the attribute (line 8). If the expression does exist, however, it generates a generalized transformation based on the context of the expression. In both cases, the salience of attribute transformations must be slightly lower than that of the rest of the level transformations, since the annotation itself must be added first before an attribute can be added.

ProgramTransformations

The final algorithm, $ProgramTransformations$, takes two decomposed representative examples, P and P', as well as a set of expressions, X, in the prior representative example, and returns the set T_{all} of inferred program transformations. First, $ProgramTransformations$ builds the sets of fundamental components, S_{all}, R_{all}, Q_{all}, and Y_{all}, for every level (lines 1-11). Then, it builds the set of context constraints (if any) for every expression (lines 14-16). Finally, $ProgramTransformations$ iterates over all levels and adds each set of level transformations to T_{all} (lines 17-19). Figure 3.17 shows the rules output by the algorithm to transform the EJB example used through this section.

3.5 Evaluation

We evaluated our approach on two criteria. First, we measured the accuracy of our inference algorithm by applying it to seven different refactoring scenarios. Second, we demonstrated the effectiveness of the automatically inferred refactorings by upgrading the testing portion of four well-known, open-source projects. The results of our evaluation show that our approach produces highly-accurate refactorings which, with few minor refinements, can be automatically-applied to large-scale applications, effectively solving the Vendor and Version Lock-in anti-patterns for applications that use annotation-based frameworks.

3.5.1 Inferred Refactorings

For our approach to be viable in a realistic setting, it has to infer refactorings with a high degree of accuracy. To assess the accuracy of our inferencing algorithm, we manually evaluated each inferred refactoring in four categories: constraints, restructurings, rule execution order, and manual refinements required. The metrics used to evaluate each category are defined below. To help explain the metrics, we give examples from the transformation rules listed in Figure 3.5.

Constraint Metrics

For constraints, we measured the number of correct, excessive, erroneous, and missing constraints, defined as follows:

- *Correct.* A correct constraint accurately captures a single requirement of a transformation rule. For example, *Name matches "test.*"* correctly requires the name of any test method to start with the `test` prefix.

- *Excessive.* An excessive constraint unnecessarily limits the scope of a transformation rule. For example, *Visibility is protected* limits the applicability of rules 2 and 3 to only `protected` methods, even though `public` methods should be captured as well.

- *Erroneous.* An erroneous constraint captures requirements that are incorrect for a specific transformation rule. For example, *Has 1 parameter* would be erroneous for identifying a `setUp` method in rule 2.

- *Missing.* A missing constraint is a necessary requirement not present in a transformation rule. For example, if *Return type is "void"* were not present in rule 2, it would be a missing constraint.

Restructuring Metrics

For restructurings, we measured the number of correct, erroneous, and missing restructurings as follows:

- *Correct.* A correct restructuring performs a required atomic transformation in a transformation rule. For example, *Set access level of $method to public* in rule 2 correctly changes the visibility of a protected `setUp` method.

- *Erroneous.* An erroneous restructuring performs an atomic transformation that invalidates a transformation rule. For example, *Set scope level of $method to static* would be erroneous for rule 2 because the method must maintain its member scope.

- *Missing.* A missing restructuring is an atomic transformation not present in a transformation rule. For example, if *Add annotation "Before" to $method* were not present in rule 2, the composite refactoring would be incomplete.

Rule Order Metrics

For transformation rules, we measured the number of correct and erroneous rule execution orders (i.e., salience values) as follows:

- *Correct.* A correctly ordered transformation rule does not invalidate other rules when executed. For example, executing rule 2 before rule 1 will not preclude rule 1 from executing.
- *Erroneous.* An erroneously ordered transformation rule invalidates another rule when executed. For example, executing rule 1 before rule 2 would invalidate rule 2 by prematurely removing the super class from the target test case.

Manual Refinement Metrics

To measure the manual effort required by a developer, we counted the number of minor and major refinements needed for each refactoring as follows:

- *Minor.* A minor or small refinement is a required change to a single constraint, restructuring, or rule execution order. For example, changing *Visibility is protected* to *Visibility is at least protected* in rule 2 is a minor refinement.
- *Major.* A major or large refinement is a required addition or removal of an entire rule. For example, if rule 1 had not been inferred, a major refinement would have been needed to add it by hand.

The above criteria were used to measure the accuracy of the inferred refactorings for seven scenarios, shown in Table 3.2 with their corresponding upgrade patterns. The first two refactoring scenarios focus on upgrading from JUnit 3 to JUnit 4, as has been discussed

Scenario	Upgrade Pattern
JUnit 3 test cases to JUnit 4	Bottom-Up
JUnit 3 test suites to JUnit 4	Top-Down
JUnit 3 test cases to TestNG	Bottom-Up
JUnit 4 test cases to TestNG	Identity
Serializable classes to JDO	Bottom-Up
Serializable classes to JPA	Bottom-Up
JDO classes to JPA	Identity

Table 3.2: The seven different upgrading scenarios and their corresponding upgrade patterns.

throughout the chapter. The third and fourth scenarios focus on switching unit testing framework vendors from JUnit to TestNG [10]. The TestNG framework was developed as a next generation testing framework extending beyond unit testing to support regression, integration, and functional testing. Recognizing that many existing applications have implemented their testing functionality using JUnit, TestNG provides a hand-written automatic upgrade utility in their Eclipse plug-in. However, as of the latest version of TestNG, this upgrade utility has several software defects when upgrading JUnit 4 test cases, such as not properly removing JUnit annotations, inserting deprecated TestNG annotations, and incorrectly matching test method names. For this scenario, we used a representative example of a TestNG test case that was identical to the JUnit 4 test case example in Figure 3.1, but with the corresponding TestNG annotations.

The remaining three scenarios focus on upgrading applications to use different enterprise *orthogonal persistence* frameworks. In the first of these three scenarios, a `Serializable` class must be upgraded to use Apache's Java Data Objects (JDO) [82] annotations. In the second scenario, the same `Serializable` class must be upgraded to use the standardized J2EE Java Persistence API (JPA) [30] annotations. In the third scenario, a class marked with JDO annotations must be transitioned to use JPA annotations.[3]

[3]For the three orthogonal persistence scenarios, we have selected the subset of functionality that we have found useful in our own software development practices rather than the entire JDO and JPA specifications. Detailed representative examples for the three persistence frameworks can be found in the technical report

Scenario		Constraints				Restructurings		
Target	Upgrade	C	M	X	E	C	M	E
Test Cases	JUnit 3 to 4	29	0	5	0	11	0	0
	JUnit 3 to TestNG	28	0	5	0	10	0	0
	JUnit 4 to TestNG	25	0	0	0	11	0	0
Test Suites	JUnit 3 to 4	52	0	0	0	11	0	0
Persistence	Serializable to JDO	78	0	0	0	14	0	0
	Serializable to JPA	78	0	0	0	14	0	0
	JDO to JPA	67	0	0	0	24	0	0
Total		357	0	10	0	95	0	0

Table 3.3: The accuracy of the inferred rules for the seven different upgrading scenarios. C=Correct; M=Missing; X=Excessive; E=Erroneous; S=Small(Minor); L=Large(Major).

Scenario		Rules		Refinements	
Target	Upgrade	C	E	S	L
Test Cases	JUnit 3 to 4	5	0	5	0
	JUnit 3 to TestNG	5	0	5	0
	JUnit 4 to TestNG	5	0	0	0
Test Suites	JUnit 3 to 4	6	0	0	0
Persistence	Serializable to JDO	8	1	1	0
	Serializable to JPA	8	1	1	0
	JDO to JPA	8	1	1	0
Total		43	3	13	0

Table 3.4: The accuracy of the inferred rules for the seven different upgrading scenarios. C=Correct; M=Missing; X=Excessive; E=Erroneous; S=Small(Minor); L=Large(Major).

Tables 3.3 and 3.4 show the accuracy of the inferred refactorings. As proved in Lemmas 1-3, our algorithm does not miss any required constraints or restructurings, nor does it infer any erroneous ones. Similarly, no major refactoring refinements are required by the developer in any of the seven scenarios.

The inference algorithm generated the same five excessive constraints in both of the JUnit 3 test case upgrading scenarios. Two of these excessive constraints unnecessarily limit the applicability of the transformation rules for the `setUp` and `tearDown` methods to only protected visibility, even though such methods can be public. The remaining three excessive constraints unnecessarily require the declaring class of `setUp`, `tearDown`, and `test` methods to directly extend `TestCase`, even though an indirect extension is valid. These two situations arise due to the small sample size used by our approach (i.e., only two representative examples). Thus, while the inferred JUnit 3 test case refactorings are correct for the given examples in both scenarios, five minor refinements are necessary to make the refactorings general enough to fully capture all valid test cases.

For the persistence scenarios, each generated refactoring requires one transformation rule reordering. The inferred rules find a naming convention for both the primary database key and persisted elements in a class, however the latter is a more general version of the former (i.e., both conventions will match the primary database key). Thus, to ensure that the primary database key is annotated before regularly persisted elements, it must be given a slightly higher precedence in the execution order, resulting in one required minor refinement for each of the persistence refactorings.

The accuracy metrics presented above show that the rules automatically inferred by our algorithm have a high degree of accuracy. On average, 97% of the total constraints, restructurings, and rule orderings require no refinement by the developer, with five out of seven

---

version of this chapter [94].

Application	Lines of Code	Test Cases	Tests	`setUp`	`tearDown`
JHotDraw 7.0.9	378	2	42	2	2
JBoss Drools 4.0.3	16,942	101	453	38	11
Apache Ant 1.7.0	21,969	251	1,714	187	99
JFreeChart 1.0.8	41,056	318	1,739	39	1
Total	80,345	672	3,948	266	113

Table 3.5: Upgrade statistics for four real-world case studies.

refactoring scenarios requiring at most one minor change. The accuracy of the inferred refactorings has an important practical significance. Unlike the hand-written upgrade utility provided by TestNG, the JUnit 4 to TestNG refactorings inferred by our approach accurately upgrade all JUnit 4 test cases without requiring any manual refinement.

3.5.2 Case Studies

To demonstrate the effectiveness of a refined refactoring, we have upgraded the JUnit 3 test cases of four open-source, real-world applications to JUnit 4. Table 3.5 presents the total number of lines of testing code, `TestCase` classes, `test` methods, `setUp` methods, and `tearDown` methods upgraded in each application. Overall, we have successfully upgraded more than 80K lines of Java source code, eliminating the need to perform this refactoring by hand.

3.6 Future Work

Currently, our inferencing algorithm supports only Java 5 annotations. In the future, we plan to extend this work to support the upcoming Java 7 annotations that enable annotating a broader set of program elements. However, our structural differencing algorithm will need to be extended to handle annotated local variables in method bodies, possibly requiring static

analysis.

Many frameworks that used XML-based metadata rather than type and naming requirements in their previous versions have since transitioned to annotations. We plan to extend our approach to automatically infer refactorings for legacy applications that use XML-based metadata frameworks. Such an extension could potentially be enabled by extending the notion of a prior representative example to include XML snippets, and subsequently incorporating XML analysis into our algorithm.

Finally, finding a more unified approach to inferring refactoring rules has great potential benefits. This may be realized either by replacing the current pattern-based approach with a more sophisticated analysis, introducing a step for inputting domain-specific knowledge, or adding algorithmic support for multiple representative examples. These enhancements could eliminate the need for a developer to decide on which upgrade pattern to use, flattening the learning curve for using our system and increasing the possibility of widespread adoption.

3.7 Conclusions

This chapter presented Annotation Refactoring and Rosemari, an approach to solving the Vendor and Version lock-in problems associated with annotation-based frameworks. Our approach is based on the wide availability of upgrade guides containing examples of legacy classes and their upgraded versions. Leveraging these examples, Rosemari enables framework developers to generate refactoring utilities capable of automatically upgrading legacy applications. As demonstrated by our case studies, the inferred refactorings are highly accurate and can effectively upgrade large-scale, real-world applications.

```
rule "#1) Transform classes matching ATest"
  no-loop
  salience 10
    when
      $class : Application Class
        - Visibility is public
        - Superclass is "junit.framework.TestCase"
    then
      Remove superclass from $class
      Update $class
end
rule "#3) Transform all methods matching tearDown"
  no-loop
  salience 20
  when
    $class : Application Class
      - Visibility is public
      - Superclass is "junit.framework.TestCase"
    $method : Application Method
      - Name matches "tearDown"
      - Declaring class is $class
      - Not annotated with "After"
      - Visibility is protected
      - Scope level is member
      - Return type is "void"
      - Has 0 parameters
  then
    Add annotation "After" to $method
    Set access level of $method to public
    Update $class
    Update $method
end
rule "#5) Manage imports"
  no-loop
  salience 0
  when
    $file : Application File
      - Does not import "org.junit.After"
      - Does not import "org.junit.Before"
      - Does not import "org.junit.Test"
      - Does not import "org.junit.Assert.*"
      - Does import "junit.framework.TestCase"
  then
    Add import "org.junit.After" to $file
    Add import "org.junit.Before" to $file
    Add import "org.junit.Test" to $file
    Add static import "org.junit.Assert.*" to $file
    Remove import "junit.framework.TestCase" from $file
    Update $file
end
```

```
rule "#2) Transform all methods matching setUp"
  no-loop
  salience 20
  when
    $class : Application Class
      - Visibility is public
      - Superclass is "junit.framework.TestCase"
    $method : Application Method
      - Name matches "setUp"
      - Declaring class is $class
      - Not annotated with "Before"
      - Visibility is protected
      - Scope level is member
      - Return type is "void"
      - Has 0 parameters
  then
    Add annotation "Before" to $method
    Set access level of $method to public
    Update $class
    Update $method
end
rule "#4) Transform all methods matching test.*"
  no-loop
  salience 20
  when
    $class : Application Class
      - Visibility is public
      - Superclass is "junit.framework.TestCase"
    $method : Application Method
      - Name matches "test.*"
      - Declaring class is $class
      - Not annotated with "Test"
      - Visibility is public
      - Scope level is member
      - Return type is "void"
      - Has 0 parameters
  then
    Add annotation "Test" to $method
    Update $class
    Update $method
end
```

Figure 3.5: The generated transformation rules to refactor a JUnit 3 test case class to use JUnit 4.

EJB 2	EJB 3

`public class HelloWorldBean`
`        implements SessionBean`
`{`
`  public void ejbActivate() {}`
 	
`  public void ejbRemove() {}`
`}`
	`@Stateful`
	`public class HelloWorldBean`
	`{`
	`  @PostActivate`
	`  public void ejbActivate() {}`
	
	`  @Remove`
	`  public void ejbRemove() {}`
	`}`

Figure 3.6: A simple Enterprise Java Bean example in EJB 2 and 3.

$$
\begin{aligned}
L_{Class} &= \{Annotations, Visibility, SuperClass, SuperInterfaces\} \\
L_{Iface} &= \{Annotations, Visibility, SuperInterfaces\} \\
L_{Field} &= \{Annotations, Visibility, Scope, Type\} \\
L_{Meth} &= \{Annotations, Visibility, Scope, \\
&\quad ReturnType, Parameters, Exceptions\} \\
L_{Cons} &= \{Annotations, Visibility, Scope, Parameters, Exceptions\} \\
L_{MethInvoke} &= \{Name, Arguments\} \\
L_{ConsInvoke} &= \{Type, Arguments\} \\
L_{Arg} &= \{Expression\}
\end{aligned}
$$

Figure 3.7: The levels at which each representative example is examined.

Bottom-Up

$L_{Arg} <_L \{L_{ConsInvoke}, L_{MethInvoke}\} <_L \{L_{Cons}, L_{Meth}, L_{Field}\} <_L \{L_{Iface}, L_{Class}\}$

Top-Down

$L_{Arg} <_L \{L_{ConsInvoke}, L_{MethInvoke}\} <_L \{L_{Iface}, L_{Class}\} <_L \{L_{Cons}, L_{Meth}, L_{Field}\}$

Identity

$\{L_{Iface}, L_{Class}, L_{Cons}, L_{Meth}, L_{Field}, L_{ConsInvoke}, L_{MethInvoke}, L_{Arg}\}$

Figure 3.8: Upgrade patterns as partial orders.

$$EJB2_{HelloWorldBean} = \{V^{public}, I^{SessionBean}\}$$
$$EJB3_{HelloWorldBean} = \{A^{Stateful}, V^{public}\}$$

$$EJB2_{ejbActivate} = \{V^{public}, S^{member}, R^{void}\}$$
$$EJB3_{ejbActivate} = \{A^{PostActivate}, V^{public}, S^{member}, R^{void}\}$$

$$EJB2_{ejbRemove} = \{V^{public}, S^{member}, R^{void}\}$$
$$EJB3_{ejbRemove} = \{A^{Remove}, V^{public}, S^{member}, R^{void}\}$$

Figure 3.9: The level decomposition of the EJB example.

ALGORITHM: Level Restructurings
INPUT: Two levels $L = \{e_1, e_2, \ldots, e_m\}, L' = \{e'_1, e'_2, \ldots, e'_n\}$ where L' is the restructured version of L.
OUTPUT: A set $R = \{r_1, r_2, \ldots, r_p\}$ of restructurings.

1. $R \leftarrow \emptyset$
2. $POSELS \leftarrow L' - L$
3. For $i \leftarrow 1$ *to* $|POSELS|$
4. $\quad R \leftarrow R \bigcup$ PositiveRestructuring($POSELS[i]$)
5. $NEGELS \leftarrow L - L'$
6. Prune($NEGELS$) // remove unnecessary elements
7. For $j \leftarrow 1$ *to* $|NEGELS|$
8. $\quad R \leftarrow R \bigcup$ NegativeRestructuring($NEGELS[j]$)
9. return R

Figure 3.10: The *LevelRestructurings* algorithm for calculating the required set of restructurings to transform between two levels.

ALGORITHM: Level Constraints
INPUT: Two sets $L = \{e_1, e_2, \ldots, e_m\}, L' = \{e'_1, e'_2, \ldots, e'_n\}$ of signature elements where L' is the restructured version of L.
OUTPUT: A set $S = \{s_1, s_2, \ldots, s_q\}$ of context-free constraints.

1. $S \leftarrow \emptyset$
2. For $i \leftarrow 1$ *to* $|L|$
3. $\quad S \leftarrow S \bigcup$ PositiveConstraint($L[i]$)
4. $NEGELS \leftarrow L' - L$
5. For $j \leftarrow 1$ *to* $|NEGELS|$
6. $\quad S \leftarrow S \bigcup$ NegativeConstraint($NEGELS[j]$)
7. return S

Figure 3.11: The *LevelConstraints* algorithm for calculating the context-free set of constraints for a level.

ALGORITHM: Naming Convention
INPUT: A set $N = \{n_1, n_2, \ldots, n_m\}$ of member names
OUTPUT: A regular expression naming convention

1. $TOKENS \leftarrow$Tokenize(n_1)
2. For $i \leftarrow 2$ *to* m
3. $\quad TOKENS \leftarrow TOKENS \bigcap$ Tokenize(n_i)
4. $\quad$ If $TOKENS = \emptyset$
5. $\qquad$ return LiteralRegex(N)
6. return GeneralizedRegex($TOKENS$)

Figure 3.12: The *NamingConvention* algorithm for generalizing a set of names to a naming convention.

ALGORITHM: Context Constraints
INPUT: A set $P = \{L_1, L_2, \ldots, L_n\}$ of levels.
A set $S_{all} = \{S_1, S_2, \ldots, S_n\}$ of sets of constraints such that $\forall S_i \in S_{all}, \forall L_i \in P$, S_i is the set of level constraints for L_i.
A partial order $<_L$ on P.
OUTPUT: A set $Q_{all} = \{Q_1, Q_2, \ldots, Q_n\}$ of context constraints such that $\forall Q_i \in Q_{all}, Q_i$ is the set of constraints defining the context of L_i.

1. $Q_{all} \leftarrow \emptyset$
2. For $i \leftarrow 1$ *to* n
3. $\quad Q_i \leftarrow \emptyset$
4. $\quad$ For $j \leftarrow 1$ *to* n
5. $\quad\quad$ If $P[i] <_L P[j]$
6. $\quad\quad\quad Q_i \leftarrow Qi \bigcup S_{all}[j]$
7. $\quad Q_{all} \leftarrow Q_{all} \bigcup \{Q_i\}$
8. return Q_{all}

Figure 3.13: The *ContextConstraints* algorithm for calculating the additional constraints for every level.

ALGORITHM: Salience Values
INPUT: A set $P = \{L_1, L_2, \ldots, L_n\}$ of levels
A partial order $<_L$ on L.
OUTPUT: A set $Y_{all} = \{y_1, y_2, \ldots, y_n\}$ of salience values, $\forall\ y_i \in Y_{all}$, y_i is the salience of L_i.

1. $Y_{all} \leftarrow \emptyset$
2. For $u \leftarrow 1$ *to* n
3. $\quad y_u \leftarrow 0$
4. $\quad$ For $v \leftarrow 1$ *to* n
5. $\quad\quad$ If $P[v]\ <_L\ P[u]$
6. $\quad\quad\quad y_u \leftarrow y_u + 10$
7. $\quad Y_{all} \leftarrow Y_{all} \bigcup \{y_u\}$
8. return Y_{all}

Figure 3.14: The $SalienceValues$ algorithm for calculating the order of execution for every level.

ALGORITHM: Level Transformations
INPUT: A set $S = \{s_1, s_2, \ldots, s_k\}$ of context-free constraints.
A set $Q = \{q_1, q_2, \ldots, q_m\}$ of context constraints.
A set $R = \{r_1, r_2, \ldots, r_n\}$ of restructurings.
A set $X = \{x_1, x_2, \ldots, x_u\}$ of expression nodes.
A set $Q_X = \{Q_1, Q_2, \ldots, Q_u\}$ of sets of context constraints for the expression nodes.
A salience score Y for this level.
OUTPUT: A set $T_{level} = \{t_1, t_2, \ldots, t_p\}$ of transformations for this level.

1. $T_{level} \leftarrow \emptyset$
2. For $i \leftarrow 1$ *to* n
3. $\quad r_i \leftarrow R[i]$
4. $\quad$ If $TypeOf(r_1) =$ AttributeAddition and $|r_i.args| > 0$
5. $\quad\quad$ For $j \leftarrow 1$ *to* $|r_i.args|$
6. $\quad\quad\quad X_a \leftarrow X \bigcap r_i.args[j]$
7. $\quad\quad\quad$ If $X_a = \emptyset$
8. $\quad\quad\quad\quad T_{level} \leftarrow T_{level} \bigcup \{\text{LiteralTransformation}(r_i.args[j], Y-1)\}$
9. $\quad\quad\quad$ Else
10. $\quad\quad\quad\quad T_{level} \leftarrow T_{level} \bigcup \{\text{GeneralTransformation}(r_i.args[j], Q_a, Y-1)\}$
11. $\quad T_{level} \leftarrow T_{level} \bigcup \{\text{GeneralTransformation}(r_i, S \bigcup Q, Y)\}$
12. return T_{level}

Figure 3.15: The $LevelTransformations$ algorithm for generating a set of transformation rules for a level.

ALGORITHM: Program Transformations
INPUT: A set $P = \{L_1, L_2, \ldots, L_n\}$ of prior levels.
A set $P' = \{L'_1, L'_2, \ldots, L'_n\})$ of posterior levels.
A set $X = \{x_1, x_2, \ldots, x_m\}$ of expression nodes in an AST.
A partial order $<_L$ on L.
OUTPUT: A set $T_{all} = \{T_1, T_2, \ldots, T_q\}$ of transformations for this program.

1. $S_{all} \leftarrow \emptyset$
2. $R_{all} \leftarrow \emptyset$
3. For $i \leftarrow 1$ *to* n
4. $\quad L_1 \leftarrow P[i]$
5. $\quad L'_1 \leftarrow P'[i]$
6. $\quad S_i \leftarrow$ LevelConstraints(L_i, L'_i)
7. $\quad R_i \leftarrow$ LevelRestructurings(L_i, L'_i)
8. $\quad S_{all} \leftarrow S_{all} \bigcup \{S_i\}$
9. $\quad R_{all} \leftarrow R_{all} \bigcup \{R_i\}$
10. $Q_{all} \leftarrow$ ContextConstraints$(S_{all}, <_L)$
11. $Y_{all} \leftarrow$ SalienceValues$(P[0])$
12. $T_{all} \leftarrow \emptyset$
13. $Q_X \leftarrow \emptyset$
14. For $i \leftarrow 1$ *to* m
15. $\quad v \leftarrow V[i]$
16. $\quad Q_X \leftarrow Q_X \bigcup \{Q_{all}.v\}$
17. For $i \leftarrow 1$ *to* n
18. $\quad T_i \leftarrow$ LevelTransformations$(S_{all}[i], Q_{all}[i], R_{all}[i], X, Q_X)$
19. $\quad T_{all} \leftarrow T_{all} \bigcup T_i$
20. return T_{all}

Figure 3.16: The *ProgramTransformations* algorithm for inferring a set of transformation rules from two representative examples.

```
rule "#1) Transform classes matching HelloWorldBean"
  no-loop
  salience 0
    when
      $class : Application Class
        - Visibility is public
        - Implements "javax.ejb.SessionBean"
    then
      Remove interface from $class
      Update $class
end
rule "#2) Transform all methods matching ejbActivate"
  no-loop
  salience 10
  when
    $class : Application Class
      - Visibility is public
      - Implements "javax.ejb.SessionBean"
    $method : Application Method
      - Name matches "ejbActivate"
      - Declaring class is $class
      - Not annotated with "javax.ejb.PostActivate"
      - Visibility is public
      - Scope level is member
      - Return type is "void"
      - Has 0 parameters
  then
    Add annotation "javax.ejb.PostActivate" to $method
    Update $class
    Update $method
end
rule "#3) Transform all methods matching ejbRemove"
  no-loop
  salience 10
  when
    $class : Application Class
      - Visibility is public
      - Implements "javax.ejb.SessionBean"
    $method : Application Method
      - Name matches "ejbRemove"
      - Declaring class is $class
      - Not annotated with "javax.ejb.Remove"
      - Visibility is public
      - Scope level is member
      - Return type is "void"
      - Has 0 parameters
  then
    Add annotation "javax.ejb.Remove" to $method
    Update $class
    Update $method
end
```

Figure 3.17: The rules learned to transform the EJB example from version 2 to version 3.

Chapter 4

Related Work

4.1 Introduction

This chapter is divided into the literature related to the two different approaches developed for this thesis: *MPI Serializer* and *Rosemari*.

4.2 MPI Serializer

A wealth of existing research literature deals with some aspect of marshaling program state. As described in section 2.2.2, existing approaches for automated marshaling C++ data structures to MPI buffers do not meet all the ancillary objectives of the HPC domain. This section examines indirectly related research on implementing and improving serialization in Java, C, and C++.

Ultimately, automatic marshaling requires the ability for an external program entity to traverse an object graph. The marshaling functionality has to be able to access all the fields

of a C++ object irrespective of their access protection level, including private and protected fields. Additionally, the marshaling functionality must be able to determine type and size information about each marshaled field. In the case of pointers and dynamically allocated structures, this information is not generally available at runtime.

The standard RunTime Type Information (RTTI) [56] of C++ is not sufficient to provide support for automatic marshaling. However, efforts have been made to provide advanced facilities for C++ that enable access to runtime type information. The most notable of which is the Metaobject Protocol (MOP) for OpenC++ [21], which enhances the language with reflective capabilities providing an effective approach to dealing with the challenges outlined above. Nevertheless, standard C++ implementations do not posses reflective capabilities. This approach would be inappropriate for our objectives, as it requires a non-standard extension of C++.

An example of a mainstream, commercial object oriented programming language with built-in reflective capabilities is Java. Ever since the capability to "pickle" the state of a Java object [79] was added to the language in the form of Java Object Serialization [88], numerous approaches for optimizing this capability have been proposed. Java reflection[87] provides capabilities for runtime inspection of Java object graphs and also for modifying object state. Many of the proposed approaches of optimizing Java serialization had the aim of making the language more amenable for HPC. The proposals for faster serialization eliminated the overheads of reflection by providing custom code for individual objects [75], using native code [64, 65], and finally using runtime code generation to specialize serialization functionality [1]. Because C++ does not have built-in reflective capabilities, none of these approaches are applicable to us. It is worth noting that the vision of Java as a language for HPC has not taken hold in mainstream high performance computing.

Remote Procedure Call (RPC) systems such as Sun RPC [89] and DCE RPC [95] provide

marshaling capability for C structures. The programmer provides an IDL specification for a remote procedure call and its parameters, and a stub compiler automatically generates marshaling and unmarshaling code. However, the marshaling/unmarshaling functionality of RPC systems is not suitable for our approach. They are not viable solutions for our domain since their target language is C rather than C++, the generated code does not use standard MPI calls, and it is not easy to specify a subset of an object state to marshal. The Flick [37] compiler simplifies several aspects of IDL specification, supports several source languages, and also applies numerous optimizations that bear similarity to the techniques of generating efficient marshaling code utilized by our approach. Despite the enhancements provided by Flick, IDL-based approaches still do not support several important C++ language features (e.g., STL containers, multiple inheritence, etc.).

CORBA [70] and DCOM [17] are object-oriented extensions of the RPC systems above. While they do provide C++ language mappings, they do not cover all of the C++ language features required by our domain. Specifically, CORBA marshaling facilities support only single class inheritance and have no notion of protected-fields [69]. Furthermore, to satisfy the programming conventions of CORBA, existing C++ code either has to be modified or special factory classes must be written [70]. Neither CORBA or DCOM marshaling provide special handling for STL container classes. DCOM is platform-dependent, being mainly supported on the Windows platform. Lastly, CORBA and DCOM require runtime libraries, which may complicate deployment.

Adaptive parameter passing [63] aims at optimizing RPC marshaling by sending a subset of an object's state graph. However, the approach does not adequately meet our design objectives since it introduces a domain specific language and does not support MPI.

4.3 Rosemari

Our approach relies on automated inference of program transformation and structural program differences. While these are broad and extensive research areas, to the best of our knowlege, none of the existing techniques in either of these areas are sufficient to automatically upgrade applications that use annotation-based frameworks.

4.3.1 Technique Classification

In a comprehensive survey, Visser [103] presents a taxonomy of program transformation systems. This taxonomy divides program transformations into two broad categories: translation and rephrasing. Translation involves transforming a program from one language to another, whereas rephrasing is concerned with program-improving transformations within the same language. Refactoring is a special subclass of rephrasing that improves the design of a program while maintaining its functionality. Renovation brings a program up to date with changed requirements, and migration ports a program from one language to another.

Our approach entails changing the application code to use a different framework. The program's semantics is preserved–the program does the same thing but using a different framework, thus classifying our approach as a refactoring. However, traditional refactoring techniques do not capture large scale changes such as the use of a different framework. Upgrading an application that uses a framework based on subtyping and naming requirements to an annotation-based framework can also be considered a renovation. Additionally, transforming legacy applications written in a language without annotations to a language with annotations is migration, as the different versions of a language (e.g., Java 1.4 vs. Java 1.5) can be considered as two different languages. Since semantic equality is the main goal of our transformations, we consider our approach a refactoring.

4.3.2 Program Transformation Systems

Our technique relies on rule-based program transformations to implement the automatically-inferred refactorings. Multiple program transformation systems have appeared in the research literature. A representative of a state-of-the-art general program transformation system is Stratego/XT [104], which enables a variety of transformations from both the translation and rephrasing categories. Another example of a multi-language transformation system is DMS® [7], which focuses on scalability and efficiency.

Several other transformation systems target a single language. JaTS [19] provides a Java-like syntax for specifying program transformations in a manner similar to macros. Inject/J [41] enables program transformations at a level higher than that of an abstract syntax tree (AST) by providing a meta-model that can be manipulated via a domain-specific scripting language. TXL [23] uses a first order functional programming model, allowing explicit programmer control over several phases of the parsing and rewriting process. iXJ [15] aims at providing a visual language to enable interactive program transformations. The Smalltalk Refactoring Browser [80], a key example of a successful application of program transformation in a commercial setting, uses an extended Smalltalk syntax to specify AST pattern trees. The Arcum framework [84] uses declarative pattern matching and substitution to specify crosscutting design idioms.

While these systems are extremely powerful tools for implementing program transformations, they do not provide support for automatically inferring transformation rules, as required by our approach. However, our algorithm is general enough that it could be used to infer transformation rules in any of the above systems that support metadata transformations.

4.3.3 Program Differencing

To infer the necessary set of transformations, our technique requires the ability to calculate differences between two program versions. Program differencing is an active research area and several differencing algorithms have been proposed recently.

Dmitriev describes a make utility for Java [34] that leverages program change history to selectively recompile dependent source files. UMLDiff [106, 109] detects structural differences between two successive version of a program and accurately models the design evolution of the system. Kim et al. DSMDiff [62] identifies differences between domain-specific models. [54] use a string similarity measure to infer structural changes at or above the level of a method header, represented as first-order relational logic rules. Since none of these techniques extend beyond the method header level, they cannot be leveraged to detect upgrade patterns at the required level of granularity.

The Breakaway tool [26] helps determine the detailed correspondences between two classes through the visualization of similarities between two ASTs. The Change Distilling algorithm [38] uses an optimized version of a tree differencing algorithm for hierarchically structured data [20] to extract fine-grained source code changes. The JDiff [2] algorithm uses an augmented representation of a control-flow graph to identify changes in the behaviors between two versions of an object-oriented program. Since none of these algorithms can generalize the inferred differences, they cannot be leveraged to infer generalized refactoring rules.

4.3.4 API Evolution

Transitioning a legacy application from a convention-based to an annotation-based framework is closely-tied to the problem of API evolution, which has been a highly-active area of recent research. Explicit documentation (e.g., change annotations [22], refactoring tags [81],

metapatterns [97], and deprecation inlining [74]) has been proposed as a means of facilitating evolution of framework dependent applications.

More recent approaches aim at automating the inference and application of refactorings. CatchUp! [45] records refactorings done by framework developers and provides facilities for replaying them on the client to update application code. Extension rules [24, 25] enable generalization transformations that add variability and flexibility into the class structure of a framework, thereby enusuring consistency with client applications. RefactoringCrawler [31] combines syntactic and semantic analyses to detect refactorings in evolving components. RefacLib [91] follows a similar approach, but replaces semantic analysis with various analysis heuristics. MolhadoRef [33] is a software configuration management system that reduces merge conflicts and facilitates program evolution comprehension by tracking refactorings and being aware of program entities. An approach that automatically creates an adaptation layer to ensure binary compatability between different framework releases and client programs is presented in [27]. Diff-CatchUp [108] leverages design differences inferred by the UMLDiff algorithm described above to apply a set of heuristics to suggest API replacements in response to compilation errors.

While these approaches have all been very effective for their target domains, they differ from our approach. Specifically, they only support simple refactorings such as *Change Signature*, and they do not combine and generalize these refactorings, as required for upgrading applications that use annotation-based frameworks.

4.4 Programming by Demonstration

Inferring a set of rules from a pair of examples bears similarity to programming by demonstration [60]. For a system to be classified as “programming by demonstration,” it must meet

two criteria. First, the programmer must create the application via the same commands or process that would be used to perform the task manually. Second, the programmer must write the program by giving an example of the desired behavior. Since Rosemari enables programmers to use the standard Eclipse interface to input representative examples, it could be classified as a *refactoring by demonstration* system.

Chapter 5

Thesis Contributions and Conclusions

5.1 Thesis Contributions

This thesis has explored methodologies, techniques, and approaches for automating adaptive maintenance tasks. We have discussed the motivation and design of two approaches, MPI Serializer and Rosemari, to automating such tasks. We next reiterate and expand the descriptions of the main conceptual contributions of this thesis.

1. *A common methodology for designing automated approaches to adaptive maintenance.* We have shown that despite the sharp contrast between the HPC community's affinity for efficiency before elegance, and the enterprise community's inverse preference, a common methodology can be followed to automate these tasks. This methodology combines automated techniques, such as compiler tools; high-level specifications, such as visual models and representative examples; and synthesis of low-level code, such as object graph traversals and program transformation rules. The tools designed following the methodology have successfully automated tedious and error-prone adaptive

maintenance tasks.

2. *A novel approach to automating the generation of efficient marshaling logic for HPC applications from a high-level visual model.* Our approach, reified in the MPI Serializer tool, enables the HPC developer to select the fields of a C++ class to be marshaled. From this high-level model, efficient code is then generated to traverse the object graph at run-time and pack it into a buffer using standard MPI calls.

3. *A novel approach to automatically upgrading legacy enterprise applications to use annotation-based frameworks.* Our approach, reified in the Rosemari tool, enables a developer to specify upgrade rules using a single example of a Java class before and after it is upgraded to the latest annotation-based framework version or vendor. Based on this example, our algorithm infers a set of general refactorings that parameterizes our transformation engine which is then capable of automatically upgrading legacy applications.

5.2 Conclusions

This thesis has presented research that is concerned with automating adaptive maintenance. The goal of this research was to demonstrate that it is possible to effectively automate tedious and error-prone adaptive maintenance tasks in a diverse set of domains by exploiting high-level specifications to synthesize specialized low-level code. We believe that the insights and approaches produced by this research will be valuable in the design of future software maintenance tools and systems.

Automated approaches to adaptive maintenance have the potential to alleviate a significant burden placed on software developers. With the size and complexity of modern applications growing every day, maintenance is imposing an escalating cost on software projects. Tight

deadlines, fierce competition, and rapidly changing requirements often leave the developer with little time to familiarize themselves with new technologies required for a particular adaptive maintenance task. By helping to automating these tasks, developer efforts can be focused more on objectives and less on implementation, reducing the need for fine-grained knowledge of the new technologies and increasing productivity. This makes automated adaptive maintenance an area of vital importance both presently, and in the future. The research presented by this thesis contributes to the ultimate aspiration of fully automated adaptive maintenance.

Bibliography

[1] B. Aktemur, J. Jones, S. Kamin, and L. Clausen. Optimizing marshalling by run-time program generation. In *4th International Conference on Generative Programming And Component Engineering (GPCE'05)*, Tallinn, Estonia, October 2005.

[2] T. Apiwattanapong, A. Orso, and M. J. Harrold. A differencing algorithm for object-oriented programs. In *Automated Software Engineering, 2004. Proceedings.* 19th *International Conference on*, pages 2–13, 2004.

[3] Jeremy S. Archuleta, Eli Tilevich, and Wu-chun Feng. A Maintainable Software Architecture for Fast and Modular Bioinformatics Sequence Search. In *23rd IEEE International Conference on Software Maintenance*, Paris, France, October 2007.

[4] Ittai Balaban, Frank Tip, and Robert Fuhrer. Refactoring support for class library migration. In *OOPSLA '05: Proceedings of the* 20th *Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 265–279, New York, NY, USA, 2005. ACM.

[5] N.A. Barricelli. Esempi Numerici di Processi di Evoluzione. *Methodos*, 6(21-22):45–68, 1954.

[6] J.J. Barton and L.R. Nackman. *Scientific and Engineering C++: An Introduction with Advanced Techniques and Examples*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1994.

[7] Ira D. Baxter, Christopher Pidgeon, and Michael Mehlich. DMS: Program transformations for practical scalable software evolution. In *ICSE '04: Proceeding of the* 26th *International Conference on Software Engineering*, pages 625–634, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

[8] K. Beck and E. Gamma. Test Infected: Programmers love writing tests. *Java Report*, 3(7):37–50, 1998.

[9] D.M. Berry. *Academic Legitimacy of the Software Engineering Discipline*. Carnegie-Mellon University, Software Engineering Institute, 1992.

[10] Cedric Beust and Hani Suleiman. *Next Generation Java Testing: TestNG and Advanced Concepts*. Addison-Wesley Professional, 2007.

[11] Barry W. Boehm. *Software Engineering Economics*. Prentice Hall PTR, Upper Saddle River, NJ, 1981.

[12] Marat Boshernitsan. Program manipulation via interactive transformations. In *OOPSLA '03: Companion of the* 18th *Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 120–121, New York, NY, USA, 2003. ACM.

[13] Marat Boshernitsan and Susan L. Graham. iXj: interactive source-to-source transformations for Java. In *OOPSLA '04: Companion to the* 19th *annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 212–213, New York, NY, USA, 2004. ACM.

[14] Marat Boshernitsan and Susan L. Graham. Interactive transformation of Java programs in Eclipse. In *ICSE '06: Proceeding of the* 28th *International Conference on Software Engineering*, pages 791–794, New York, NY, USA, 2006. ACM.

[15] Marat Boshernitsan, Susan L. Graham, and Marti A. Hearst. Aligning development tools with the way programmers think about code changes. In *CHI '07: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 567–576, New York, NY, USA, 2007. ACM.

[16] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.16: components for transformation systems. In *PEPM '06: Proceedings of the 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 95–99, New York, NY, USA, 2006. ACM.

[17] N. Brown and C. Kindel. *Distributed Component Object Model Protocol–DCOM/1.0*. Microsoft Corporation, November 1998.

[18] W.J. Brown, R.C. Malveau, H.W. McCormick III, and T.J. Mowbray. *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc. New York, NY, USA, 1998.

[19] F. Castor and P. Borba. A language for specifying Java transformations. In *V Brazilian Symposium on Programming Languages*, pages 236–251, 2001.

[20] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 493–504, New York, NY, USA, 1996. ACM.

[21] S. Chiba. A metaobject protocol for C++. *ACM SIGPLAN Notices*, 30(10):285–299, 1995.

[22] Kingsum Chow and David Notkin. Semi-automatic update of applications in response to library changes. In *ICSM '96: Proceedings of the 1996 International Conference on Software Maintenance*, page 359, Washington, DC, USA, 1996. IEEE Computer Society.

[23] J. Cordy. The txl source transformation language. *Science of Computer Programming*, 61(3):190–210, August 2006.

[24] Mariela Cortes, Marcus Fontoura, and Carlos Lucena. Using refactoring and unification rules to assist framework evolution. *SIGSOFT Softw. Eng. Notes*, 28(6):1–5, 2003.

[25] M.I. Cortés, M.F.M.C. Fontoura, and C.J.P. de Lucena. A Rule-based Approach to Framework Evolution. *Journal of Object Technology (JOT)*, 5(1), jan-feb 2006.

[26] Rylan Cottrell, Joseph J. C. Chang, Robert J. Walker, and Jörg Denzinger. Determining detailed structural correspondence for generalization tasks. In *ESEC-FSE '07: Proceedings of the the* 6th *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 165–174, New York, NY, USA, 2007. ACM.

[27] Ilie Şavga and Michael Rudolf. Refactoring-based support for binary compatibility in evolving frameworks. In *GPCE '07: Proceedings of the* 6th *International Conference on Generative Programming and Component Engineering*, pages 175–184, New York, NY, USA, 2007. ACM.

[28] K. Czarnecki and U.W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co. New York, NY, USA, 2000.

[29] Aaron E. Darling, Lucas Carey, and Wu-chun Feng. The design, implementation, and evaluation of mpiBLAST. In *ClusterWorld Conference & Expo and the 4th International Conference on Linux Cluster: The HPC Revolution 2003*, San Jose, California, June 2003. Best Paper: Applications Track.

[30] Linda DeMichiel and Michael Keith. JSR 220: Enterprise JavaBeans 3.0, May 2006. http://jcp.org/aboutJava/communityprocess/final/jsr220/index.html.

[31] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automated detection of refactorings in evolving components. In *ECOOP*, pages 404–428, 2006.

[32] D. Dig and R. Johnson. How do APIs evolve? a story of refactoring. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(2):83–107, 2006.

[33] Danny Dig, Kashif Manzoor, Ralph Johnson, and Tien N. Nguyen. Refactoring-aware configuration management for object-oriented programs. In *ICSE '07: Proceedings of the* 29th *International Conference on Software Engineering*, pages 427–436, Washington, DC, USA, 2007. IEEE Computer Society.

[34] Mikhail Dmitriev. Language-specific make technology for the Java programming language. *SIGPLAN Not.*, 37(11):373–385, 2002.

[35] R. Eckstein, M. Loy, and D. Wood. *Java Swing*. O'Reilly & Associates, Inc. Sebastopol, CA, USA, 1998.

[36] Eclipse Foundation. Eclipse Java development tools, March 2008. http://www.eclipse.org/jdt.

[37] Eric Eide, Kevin Frei, Bryan Ford, Jay Lepreau, and Gary Lindstrom. Flick: a flexible, optimizing idl compiler. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997*

conference on Programming language design and implementation, pages 44–56, New York, NY, USA, 1997. ACM.

[38] Beat Fluri, Michael Wuersch, Martin Pinzger, and Harald Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Trans. Softw. Eng.*, 33(11):725–743, 2007.

[39] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[40] GCC-XML, the XML output extension to GCC, 2007. http://www.gccxml.org/.

[41] Thomas Genssler and Volker Kuttruff. Source-to-source transformation in the large. In *Modular Programming Languages*, pages 254–265. Springer-Verlag, 2003.

[42] D. Goujon, M. Michel, J. Peeters, and J.E. Devaney. Automap and autolink: Tools for communicating complex and dynamic data-structures using MPI. In *Lectures Notes in Computer Science*, volume 1362, page 98, 1998. Presented at CANPC98.

[43] L.L. Gremillion. Determinants of program repair maintenance requirements. *Communications of the ACM*, 27(8):826–832, 1984.

[44] C.J. Guzak, J.L. Bogdan, G.H. Pitt III, and C.H. Chew. Tree view control, November 1999. US Patent 5,977,971.

[45] Johannes Henkel and Amer Diwan. CatchUp!: capturing and replaying refactorings to support API evolution. In *ICSE '05: Proceedings of the* 27th *International Conference on Software Engineering*, pages 274–283, New York, NY, USA, 2005. ACM.

[46] R. Hillson and M. Iglewski. C++2MPI: A software tool for automatically generating MPI datatypes from C++ classes. In *International Conference on Parallel Computing*

in Electrical Engineering (PARELEC'00), Washington, DC, USA, 2000. IEEE Computer Society.

[47] L. Hochstein, J. Carver, F. Shull, S. Asgari, V. Basili, J.K. Hollingsworth, and M. Zelkowitz. HPC programmer productivity: A case study of novice HPC programmers. In *ACM/IEEE Supercomputing Conference (SC'05)*, 2005.

[48] Lorin Hochstein and Victor R. Basili. An empirical study to compare two parallel programming models. In *SPAA '06: Proceedings of the eighteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 114–114, New York, NY, USA, 2006. ACM.

[49] International Organization for Standardization, Geneva, Switzerland. *ISO/IEC 14764:2006: Software Engineering – Software Life Cycle Processes – Maintenance*, 2006.

[50] P. Kambadur, D. Gregor, A. Lumsdaine, and A. Dharurkar. Modernizing the C++ interface to MPI. In *13th European PVM/MPI Users Group Meeting (EuroPVM/MPI'06)*. Springer, 2006.

[51] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP)*, pages 327–353, London, UK, 2001. Springer-Verlag.

[52] Miryung Kim, Jonathan Beall, and David Notkin. Discovering and representing logical structure in code change. Technical report, University of Washington, 2007.

[53] Miryung Kim and David Notkin. Program element matching for multi-version program analyses. In *MSR '06: Proceedings of the 2006 International Workshop on Mining Software Repositories*, pages 58–64, New York, NY, USA, 2006. ACM.

[54] Miryung Kim, David Notkin, and Dan Grossman. Automatic inference of structural changes for matching across program versions. In *The 29th International Conference on Software Engineering (ICSE'07)*, pages 333–343, 2007.

[55] G. Kniesel, P. Costanza, and M. Austermann. Jmangler-a framework for load-time transformation of java classfiles. pages 98–108, 2001.

[56] A. Koenig. *The C++ Language Standard. Report ISO/IEC 14882: 1998*, 1998. http://www.nctis.org/cplusplus.htm.

[57] Meir M. Lehman and Laszlo A. Belady. *Program Evolution: Processes of Software Change*. Academic Press Professional, Inc., San Diego, CA, 1985.

[58] B.P. Lester. *The Art of Parallel Programming*. Prentice-Hall, Inc., Upper Saddle River, New Jersey, USA, 1993.

[59] Z. Li and Y. Zhou. Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. In *Joint proceedings of the 10th European software engineering conference and the 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 306–315. ACM, 2005.

[60] H. Lieberman. *Your Wish is My Command: Programming By Example*. Morgan Kaufmann, 2001.

[61] B.P. Lientz and E.B. Swanson. *Software Maintenance Management*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1980.

[62] Y. Lin, J. Gray, and F. Jouault. Dsmdiff: A differentiation tool for domain-specific models. *European Journal of Information Systems*, 16:349–361, 2007.

[63] C.V. Lopes. Adaptive parameter passing. In *Object Technologies for Advanced Software: Second JSSST International Symposium (ISOTAS'96)*, Kanazawa, Japan, March 1996. Springer.

[64] J. Maassen, R. Van Nieuwpoort, R. Veldema, H. Bal, T. Kielmann, C. Jacobs, and R. Hofman. Efficient Java RMI for parallel programming. *ACM Transactions on Programming Languages and Systems*, 23(6):747–775, 2001.

[65] J. Maassen, R. van Nieuwpoort, R. Veldema, H.E. Bal, and A. Plaat. An efficient implementation of Java's remote method invocation. In *The seventh ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 173–182, New York, New York, USA, 1999. ACM Press.

[66] V. Massol and T. Husted. *JUnit in Action*. Manning, 2004.

[67] U. Mello and I. Khabibrakhmanov. On the reusability and numeric efficiency of C++ packages in scientific computing. In *The ClusterWorld Conference and Expo*, pages 23–26, June 2003.

[68] Message Passing Interface Forum (MPIF). MPI-2: Extensions to the message-passing interface. Technical report, University of Tennessee, Knoxville, 1996.

[69] Object Management Group. *Objects by Value Specification*, 1998. ftp://ftp.omg.org/pub/docs/orbos/98-01-18.pdf.

[70] Object Management Group. *C++ Language Mapping Specifications*, 2003. http://www.omg.org/docs/formal/03-06-03.pdf.

[71] William F. Opdyke. *Refactoring object-oriented frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1992.

[72] Debu Panda, Doug Clarke, and Merrick Schincariol. EJB 3.0 migration. Technical report, Oracle, October 2005.

[73] R. Pawlak, C. Noguera, and N. Petitprez. Spoon: Program analysis and transformation in Java. Technical report, INRIA Research Report, 2006.

[74] Jeff H. Perkins. Automatically generating refactorings to support API evolution. In *PASTE '05: Proceedings of the* 6th *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 111–114, New York, NY, USA, 2005. ACM.

[75] M. Philippsen, B. Haumacher, and C. Nester. More efficient serialization and RMI for Java. *Concurrency Practice and Experience*, 12(7):495–518, 2000.

[76] M. Proctor, M. Neale, P. Lin, and M. Frandsen. Drools Documentation. Technical report, JBoss Inc., 2006.

[77] E. Renault and C. Parrot. MPI Pre-Processor: Generating MPI derived datatypes from C datatypes automatically. In *The 2006 International Conference Workshops on Parallel Processing*, pages 248–256, Washington, DC, USA, 2006. IEEE Computer Society.

[78] Chris Richardson. Untangling enterprise Java. *Queue*, 4(5):36–44, 2006.

[79] Roger Riggs, Jim Waldo, Ann Wollrath, and Krishna Bharat. Pickling state in the Java™system. In *COOTS'96: Proceedings of the 2nd conference on USENIX Conference on Object-Oriented Technologies (COOTS)*, pages 19–19, Berkeley, CA, USA, 1996. USENIX Association.

[80] D. Roberts and J. Brant. Tools for making impossible changes - experiences with a tool for transforming large Smalltalk programs. *Software, IEE Proceedings -*, 151(2):49–56, 2004. 1462-5970.

[81] Stefan Roock and Andreas Havenstein. Refactoring tags for automatic refactoring of framework dependent applications. In *Proc. Int'l Conf. eXtreme Programming and Flexible Processes in Software Engineering (XP)*, 2002.

[82] Craig Russell. Java Data Objects 2.1, June 2007. http://db.apache.org/jdo/specifications.html.

[83] S. Sankaranarayanan, F. Ivancic, and A. Gupta. Mining library specifications using inductive logic programming. In *ICSE '08: Proceedings of the* 30th *International Conference on Software Engineering*. ACM, 2008.

[84] Macneil Shonle, William G. Griswold, and Sorin Lerner. Beyond refactoring: a framework for modular maintenance of crosscutting design idioms. In *ESEC-FSE '07: Proceedings of the the* 6th *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 175–184, New York, NY, USA, 2007. ACM.

[85] Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, 2002.

[86] Ralf Stuckert. JUnit reloaded, December 2006. http://today.java.net/pub/a/today/2006/12/07/junit-reloaded.html.

[87] Sun Microsystems. *Java Core Reflection API and Specification*, 1997.

[88] Sun Microsystems. *Java Object Serialization Specification*, 2001.

[89] Sun Microsystems. *RPC: Remote Procedure Call Protocol Specification*, 2004.

[90] Sun Microsystems Inc. Java 2 Platform, Enterprise Edition (J2EE), 2003.

[91] Kunal Taneja, Danny Dig, and Tao Xie. Automated detection of API refactorings in libraries. In *ASE '07: Proceedings of the* 22nd *IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2007.

[92] Wesley Tansey and Eli Tilevich. Annotation refactoring: Inferring upgrade transformations for legacy applications. In *OOPSLA '08: Proceedings of the* 23rd *Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, New York, NY, USA, 2008. ACM.

[93] Wesley Tansey and Eli Tilevich. Efficient automated marshaling of C++ data structures for MPI applications. In *IPDPS '08: Proceedings of the* 22nd *IEEE International Parallel and Distributed Processing Symposium*, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

[94] Wesley Tansey and Eli Tilevich. Refactoring object-oriented applications for metadata-based frameworks. Technical report, Virginia Tech, January 2008.

[95] The Open Group. *DCE 1.1 RPC Specification*, 1997. http://www.opengroup.org/onlinepubs/009629399/.

[96] TOP500. Top 500 supercomputing sites - architecture. http://top500.org/stats/28/archtype/.

[97] Tom Tourwé and Tom Mens. Automated support for framework-based software evolution. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, page 148, Washington, DC, USA, 2003. IEEE Computer Society.

[98] Dániel Varró and Zoltán Balogh. Automating model transformation by example using inductive logic programming. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, pages 978–984, New York, NY, USA, 2007. ACM.

[99] Todd L. Veldhuizen and M. Ed Jernigan. Will C++ be faster than Fortran? In *ISCOPE '97: Proceedings of the Scientific Computing in Object-Oriented Parallel Environments*, pages 49–56, London, UK, 1997. Springer-Verlag.

[100] Donald Vines and Kevin Sutter. Migrating legacy Hibernate applications to OpenJPA and EJB 3.0, August 2007. http://www.ibm.com/developerworks/websphere/techjournal/0708_vines/0708_vines.html.

[101] Virginia Tech. Virginia Tech terascale computing facility. http://www.arc.vt.edu/arc/SystemX/.

[102] Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. In *Rewriting Techniques and Applications: 12th International Conference, RTA 2001, Utrecht, The Netherlands, May 2001, Proceedings:*, page 357. Springer-Verlag, 2001.

[103] Eelco Visser. A survey of strategies in program transformation systems. *Electronic Notes in Theoretical Computer Science*, 57, 2001.

[104] Eelco Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer et al., editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Spinger-Verlag, June 2004.

[105] N. Wilde, P. Matthews, and R. Huitt. Maintaining object-oriented software. *IEEE Software*, 10(1):75–80, 1993.

[106] Zhenchang Xing and Eleni Stroulia. UMLDiff: an algorithm for object-oriented design differencing. In *ASE '05: Proceedings of the* 20th *IEEE/ACM International Conference on Automated Software Engineering*, pages 54–65, New York, NY, USA, 2005. ACM.

[107] Zhenchang Xing and Eleni Stroulia. Refactoring practice: How it is and how it should be supported - an Eclipse case study. In *ICSM '06: Proceedings of the* 22nd *IEEE International Conference on Software Maintenance*, pages 458–468, Washington, DC, USA, 2006. IEEE Computer Society.

[108] Zhenchang Xing and Eleni Stroulia. API-evolution support with Diff-CatchUp. *IEEE Trans. Softw. Eng.*, 33(12):818–836, 2007.

[109] Zhenchang Xing and Eleni Stroulia. Differencing logical UML models. *Automated Software Engineering*, 14(2):215–259, 2007. 10.1007/s10515-007-0007-3.

Appendix A

MPI Serializer Code Examples

This appendix contains the generated code for the `Mass` example used in the MPI Serializer chapter. The generated code is mangled to prevent duplicate names from creating compile- or run-time errors. Rather than directly using literal values for the non-public field accesses, the generated code uses macro values; this decouples the isomorphic structure generator and field offset generator from the marshaling logic generator. Since an unbounded object graph subset is selected for the master to worker communication, custom traversal logic is generated. However, since a bounded subset is selected for the worker to master communication, only the logic needed to initialize an `MPI_Datatype` is generated.

A.1 Master to Worker Communication

A.1.1 Header File

```
#ifndef __MPI_SERIALIZER_H
#define __MPI_SERIALIZER_H
```

```
#include "MPI_Serializer_Defines.h"
#include "mpi.h"
#include <vector>
#include <map>
#include <set>

extern MPI_Datatype masterToWorker_Mass_surrounding_Mass_item;

enum MPI_Descriptor_masterToWorker
{
  masterToWorker
};

int MPI_Pack(Mass* inbuf, int incount, MPI_Descriptor_masterToWorker datatype,
             char *outbuf, int outcount, int *position , MPI_Comm comm);
void init ();
void pack_masterToWorker_Mass_surrounding(
   std :: vector<Mass,std:: allocator <Mass> >* toPack,
   char* buf, int size , int* position , MPI_Comm comm);
void pack_masterToWorker(Mass* toPack, char* buf, int size, int* position ,
                         MPI_Comm comm);
void unpack_masterToWorker_Mass_surrounding(
   std :: vector<Mass,std:: allocator <Mass> >* toPack,
   char* buf, int size , int* position , MPI_Comm comm);
void unpack_masterToWorker(Mass* toPack, char* buf, int size,
                           int* position , MPI_Comm comm);
```

```
#endif
```

A.1.2 CPP File

```
#include "MPI_Serializer.h"
using namespace std;

MPI_Datatype masterToWorker_Mass_surrounding_Mass_item;
int MPI_Pack(Mass* inbuf, int incount, MPI_Descriptor_masterToWorker datatype,
              char *outbuf, int outcount, int *position, MPI_Comm comm)
{
  int success = MPI_SUCCESS;
  for(int i = 0; i < incount; i++)
  {
    pack_masterToWorker(&inbuf[i], outbuf, outcount, position, comm);
  }
  return success;
}
void init()
{
  unsigned char* base;
  Mass Mass_masterToWorker_Mass_surrounding_Mass_item;
base = (unsigned char*)&Mass_masterToWorker_Mass_surrounding_Mass_item;
unsigned char* Mass_masterToWorker_Mass_surrounding_Mass_item_Mass_surrounding = base +
                                                    Mass_Mass_surrounding_offset;
unsigned char* Mass_masterToWorker_Mass_surrounding_Mass_item_Mass_force_x = base +
                                                    Mass_Mass_force_x_offset;
```

```
unsigned char* Mass_masterToWorker_Mass_surrounding_Mass_item_Mass_force_y = base +
                                                              Mass_Mass_force_y_offset ;
unsigned char* Mass_masterToWorker_Mass_surrounding_Mass_item_Mass_mass = base +
                                                              Mass_Mass_mass_offset;
unsigned char* Mass_masterToWorker_Mass_surrounding_Mass_item_Mass_x = base +
                                                              Mass_Mass_x_offset;
unsigned char* Mass_masterToWorker_Mass_surrounding_Mass_item_Mass_y = base +
                                                              Mass_Mass_y_offset;
MPI_Datatype masterToWorker_Mass_surrounding_Mass_item_types[3] =
                                    {MPI_FLOAT,MPI_FLOAT,MPI_FLOAT};
int masterToWorker_Mass_surrounding_Mass_item_counts[3] = {1,1,1};
MPI_Aint masterToWorker_Mass_surrounding_Mass_item_disps[4];
MPI_Address(
   (std :: vector<Mass >*)Mass_masterToWorker_Mass_surrounding_Mass_item_Mass_surrounding,
                 &masterToWorker_Mass_surrounding_Mass_item_disps[0]);
MPI_Address(&Mass_masterToWorker_Mass_surrounding_Mass_item.mass,
                &masterToWorker_Mass_surrounding_Mass_item_disps[1]);
MPI_Address(&Mass_masterToWorker_Mass_surrounding_Mass_item.x,
                &masterToWorker_Mass_surrounding_Mass_item_disps[2]);
MPI_Address(&Mass_masterToWorker_Mass_surrounding_Mass_item.y,
                &masterToWorker_Mass_surrounding_Mass_item_disps[3]);
for(int i = 3; i >= 0; --i)
  masterToWorker_Mass_surrounding_Mass_item_disps[i] -=
                 masterToWorker_Mass_surrounding_Mass_item_disps[0];


MPI_Type_struct(3,masterToWorker_Mass_surrounding_Mass_item_counts,
                &masterToWorker_Mass_surrounding_Mass_item_disps[1],
```

```
                    masterToWorker_Mass_surrounding_Mass_item_types,
                    &masterToWorker_Mass_surrounding_Mass_item);
MPI_Type_commit(&masterToWorker_Mass_surrounding_Mass_item);



}
void pack_masterToWorker_Mass_surrounding(
                    std :: vector<Mass,std:: allocator <Mass> >* toPack, char* buf, int size,
                    int* position , MPI_Comm comm){
unsigned int toPack_size = toPack->size();
MPI_Pack(&toPack_size, 1, MPI_UNSIGNED, buf, size, position, comm);
for(std :: vector<Mass,std:: allocator <Mass> >::iterator item = toPack->begin();
      item != toPack->end(); item++)
{
  MPI_Pack((void*)&((*item)), 1, masterToWorker_Mass_surrounding_Mass_item, buf,
              size , position , comm);
}

}

void pack_masterToWorker(Mass* toPack, char* buf, int size , int* position ,
                              MPI_Comm comm){
unsigned char* base = (unsigned char*)toPack;
unsigned char* toPack_Mass_surrounding = base + Mass_Mass_surrounding_offset;
unsigned char* toPack_Mass_force_x = base + Mass_Mass_force_x_offset;
unsigned char* toPack_Mass_force_y = base + Mass_Mass_force_y_offset;
unsigned char* toPack_Mass_mass = base + Mass_Mass_mass_offset;
```

```
unsigned char* toPack_Mass_x = base + Mass_Mass_x_offset;
unsigned char* toPack_Mass_y = base + Mass_Mass_y_offset;

pack_masterToWorker_Mass_surrounding(((std::vector<Mass >*)toPack_Mass_surrounding),
                                     buf, size, position, comm);
MPI_Pack( &((*toPack).Mass::mass), 1, MPI_FLOAT, buf, size, position, comm);
MPI_Pack( &((*toPack).Mass::x), 1, MPI_FLOAT, buf, size, position, comm);
MPI_Pack( &((*toPack).Mass::y), 1, MPI_FLOAT, buf, size, position, comm);

}

void unpack_masterToWorker_Mass_surrounding(
                     std::vector<Mass,std::allocator<Mass> >* toPack, char* buf,
                     int size, int* position, MPI_Comm comm){
toPack->clear();
unsigned int toPack_size;
MPI_Unpack( buf, size, position, &toPack_size, 1, MPI_UNSIGNED, comm );
for(unsigned int i = 0; i < toPack_size; i++)
{
  Mass item;
  MPI_Unpack( buf, size, position, &(item), 1,
           masterToWorker_Mass_surrounding_Mass_item, comm);
  toPack->push_back(item);
}

}
```

```
void unpack_masterToWorker(Mass* toPack, char* buf, int size,
                           int* position, MPI_Comm comm){
unsigned char* base = (unsigned char*)toPack;
unsigned char* toPack_Mass_surrounding = base + Mass_Mass_surrounding_offset;
unsigned char* toPack_Mass_force_x = base + Mass_Mass_force_x_offset;
unsigned char* toPack_Mass_force_y = base + Mass_Mass_force_y_offset;
unsigned char* toPack_Mass_mass = base + Mass_Mass_mass_offset;
unsigned char* toPack_Mass_x = base + Mass_Mass_x_offset;
unsigned char* toPack_Mass_y = base + Mass_Mass_y_offset;

unpack_masterToWorker_Mass_surrounding(((std::vector<Mass >*)toPack_Mass_surrounding),
                                       buf, size, position, comm);
MPI_Unpack( buf, size, position, &((*toPack).Mass::mass), 1, MPI_FLOAT, comm);
MPI_Unpack( buf, size, position, &((*toPack).Mass::x), 1, MPI_FLOAT, comm);
MPI_Unpack( buf, size, position, &((*toPack).Mass::y), 1, MPI_FLOAT, comm);

}
```

A.2 Worker to Master Communication

A.2.1 Header File

```
#ifndef __MPI_SERIALIZER_H
#define __MPI_SERIALIZER_H
#include "MPI_Serializer_Defines.h"
#include "mpi.h"
```

```
#include <vector>
#include <map>
#include <set>

extern MPI_Datatype masterToWorker;

  void init ();

#endif
```

A.2.2 CPP File

```
#include "MPI_Serializer.h"
using namespace std;

MPI_Datatype masterToWorker;
void init ()
{
  unsigned char* base;
  Mass Mass_masterToWorker;
base = (unsigned char*)&Mass_masterToWorker;
unsigned char* Mass_masterToWorker_Mass_surrounding =
                        base + Mass_Mass_surrounding_offset;
unsigned char* Mass_masterToWorker_Mass_force_x =
                        base + Mass_Mass_force_x_offset ;
unsigned char* Mass_masterToWorker_Mass_force_y =
                        base + Mass_Mass_force_y_offset ;
```

```
unsigned char* Mass_masterToWorker_Mass_mass =
                              base + Mass_Mass_mass_offset;
unsigned char* Mass_masterToWorker_Mass_x =
                              base + Mass_Mass_x_offset;
unsigned char* Mass_masterToWorker_Mass_y =
                              base + Mass_Mass_y_offset;
MPI_Datatype masterToWorker_types[2] =
                              {MPI_FLOAT,MPI_FLOAT};
int masterToWorker_counts[2] = {1,1};
MPI_Aint masterToWorker_disps[3];
MPI_Address((std::vector<Mass >*)Mass_masterToWorker_Mass_surrounding,
                              &masterToWorker_disps[0]);
MPI_Address((float*)Mass_masterToWorker_Mass_force_x,
                              &masterToWorker_disps[1]);
MPI_Address((float*)Mass_masterToWorker_Mass_force_y,
                              &masterToWorker_disps[2]);
for(int i = 2; i >= 0; --i)
  masterToWorker_disps[i] -= masterToWorker_disps[0];


MPI_Type_struct(2,masterToWorker_counts,&masterToWorker_disps[1],
                             masterToWorker_types, &masterToWorker);
MPI_Type_commit(&masterToWorker);


}
```