Concurrency Analysis of Java RMI Using Source Transformation and Verisoft

by

Tim Cassidy

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Master of Science

Queen's University
Kingston, Ontario, Canada
January 2004

Abstract

Concurrent programming has two main benefits: First, it allows natural solutions to software design problems that are inherently parallel or distributed. Second, concurrent programs offer potentially better efficiency than sequential programs. However, concurrent programming poses many challenges that do not exist in the sequential setting. For instance, the processes in the system may livelock, diverge, or even deadlock.

The Java Remote Method Invocation (RMI) package facilitates the implementation of concurrent applications in which, for instance, the processes reside on different hosts and communicate over the internet. More precisely, it hides the details of network communication. Unfortunately, it does not relieve the programmer from the potential pitfalls of controlling the concurrent access to remote objects. Consequently, RMI applications are prone to the same problems as concurrent programs in general.

To address this problem, this thesis presents an approach, named JCUV (Java to C++ Using Verisoft) that allows Java RMI programs to be analyzed with respect to deadlock, livelock, divergences, and assertion violations. The approach consists of two steps: First, the Java RMI program is translated into an equivalent C++ program using automated source code transformation. Second, the resulting C++ program is analyzed with the state space exploration tool Verisoft. My thesis discusses the details of the transformation and evaluates the approach on a number of small examples.

Acknowledgments

I would like to thank the following people for making this research possible:

- **Gordon Cassidy** - My dad has helped see me through the completion of my Master's from start to finish.
- **Jim Cordy and Thomas R. Dean** - They have given me the guidance to understand source transformation and more generally Software Engineering.
- **Juergen Dingel** - For prompting my interest in model verification and being a friend.
- **Dean Jin** - For helping me through the LaTeX learning curve.
- **Debby Robertson** - For her support and efforts in processing my late application to Queen's University - in addition to all the times she has helped me out since then.
- **Marina Zekios** - For being there for me.

In addition, I would like to thank Donald R. Taylor (Committee Chair) and Robert D. Tennent (Head's Representative) for sitting on my thesis defense committee. Lastly, I would like to thank Douglas Noel Adams for teaching me that the answer is and always has been 42.

Contents

List of Figures

Chapter 1

Introduction and Motivation

A deadlock is a state where the next instruction of every process is blocking. A livelock occurs when two processes/threads are able to change their state (i.e. are not blocked) but never make any useful progress[Inf99]. A divergence occurs whenever no communication occurs between two threads or processes after a set amount of time. All of these situations should be viewed as run-time errors and all can be found (within certain limitations) using the concurrency analysis tool Verisoft[God01].

Any message protocol that uses synchronization is vulnerable to deadlock. More commonly, these protocols are vulnerable to divergences. In multi-user environments divergences can make an application completely unusable as the number of users increases. Thus, it becomes critical to prevent both divergences and deadlocks.

Java RMI[Sun01] (Remote Method Invocation) is frequently used for the development of distributed systems involving, for instance, e-commerce applications. The appeal of Java RMI is that it frees the programmer from having to worry about the details of network communication like opening, closing and connecting to sockets.

```
public class PeerA extends UnicastRemoteObject
       implements PeerAInterface, Serializable{

  synchronized public void callBack () {
    //never make it into here
  }

  synchronized public void run () {
    try {
      String name = "PeerB";
      PeerBInterface peerB =
        (PeerBInterface) Naming.lookup(name);
      peerB.executeTask();
    }
    catch (Exception exception_){
      exception_.printStackTrace();
    }
  }

  public static void main(String args[]) {
    try {
      String name = "PeerA";
      PeerAInterface peerA = new PeerA();
      Naming.rebind(name, peerA);
      peerA.run();
    }
    catch (Exception exception_){
      exception_.printStackTrace();
    }
  }
}
```

```
public class PeerB extends UnicastRemoteObject
       implements PeerBInterface, Serializable
{

  public void executeTask(){
    try {
      String name = "PeerA";
      PeerAInterface peerA =
        (PeerAInterface) Naming.lookup(name);
      peerA.callBack();
    }
    catch (Exception exception_){
      exception_.printStackTrace();
    }
  }

  public static void main(String[] args) {
    String name = "PeerB";
    try {
      PeerB peerB = new PeerB();
      Naming.rebind(name, peerB);
    } catch (Exception exception_) {
      exception_.printStackTrace();
    }
  }
}
```

Figure 1.1: Simple RMI code example that illustrates a Java RMI program that will definitely result in deadlock.

Unfortunately, Java RMI provides little help when it comes to getting the intricate details of concurrency control right. This problem is exacerbated when the number of clients increases, because the developer may easily overlook, for instance, the introduction of a circular dependency between remote objects that could result in deadlock. Consider, for example, the RMI code in Figure 1.1 which will always result in deadlock.

In Figure 1.1, `PeerB` is started first, then once it has bound itself to the RMI registry (using the `Naming.rebind` method), `PeerA` can be started. `PeerA` binds itself to the RMI registry (by also using the `Naming.rebind` method) and then enters its run method and then locates `PeerB` using the RMI registry (by invoking `Naming.lookup`). It then invokes `PeerB`'s `executeTask` method. `PeerB`'s `executeTask` method is a

`synchronized` method, the `synchronized` keyword is used to ensure that only one thread can access the synchronized object at a time. `PeerB` then attempts to invoke `PeerA`'s `callBack` method, but is blocked since `PeerA`'s run method is still waiting for a return from `PeerB`'s `executeTask` method - deadlock!

In this thesis, I present an approach, called JCUV (Java to C++ Using Verisoft) to analyze the concurrency aspects of Java RMI programs. The approach first translates the Java program into an equivalent C++ program using automated source transformation. Once more, automated source transformation is used to generate a model of the RMI portion of the original Java program in C++. Finally, the resulting C++ program is analyzed for deadlocks, divergences and livelocks using Verisoft.

1.1 Summary

Before addressing the specifics of my thesis, Chapter two will discuss the background material that had to be researched before my implementation could be started. Chapter three will give some of the rationale for the choice of artifacts that were used in my implementation. In addition, Chapter three will also outline my solution as a whole. Chapter four will discuss the specifics of the first step of my solution. Chapter five discusses the specifics of the second step of my solution and Chapter six will discuss the third and last step of my solution. Chapter seven contains the experiments that were completed to test my implementation. My last Chapter contains the conclusions of my dissertation.

Chapter 2

Background

2.1 Difficulties in Software Engineering

Software engineering is a process that allows programmers/developers to construct correct and sound software[BD00]. However, in reality (the software industry) numerous factors inhibit this process from being completed in its entirety. All of the following work against the software engineering process to produce correct and valid software[1]:

- Moving targets (insufficient requirements analysis, users' needs/desires changing)
- Lack of time/resources (finances, personnel)
- Lack of formal process or standards - As indicated by Jones

[1]Correct and valid software is simply software that does what is intended in the manner that was intended.

> "A number of industry studies (TRW, Nippon Electric, and Mitre Corp., among others) indicate that design activities introduce between 50 and 65 percent of all errors (and ultimately, all defects) during the software process. However, formal review techniques have been shown to be up to 75 percent effective in uncovering design flaws." [Jon86]

Thus, if a formal review is undertaken, the cost of the development and maintenance phases are significantly reduced.

- Communication - Difficulties in synchronizing efforts amongst the active participants in the software engineering process
- Inadequate training

An important component to the software engineering process, at any phase/stage, is using modelling tools, sometimes referred to as Model Checking or Verification Software. A variety of modelling tools exist, such as SMV[McM00], Spin[Hol97b], Rational Quality Architect Realtime Edition[Cor02] and Verisoft[God01].

The majority of these tools construct a finite state space and explore that state space to determine if it satisfies the properties of interest, these include freedom from deadlock, livelocks, assertion violations, etc.

2.2 Model Checking Tool Problems

There are four major problems that are associated with the use of model-checking tools[HT01].

2.2.1 Model Construction Problem

Often it is difficult or potentially impossible to create a meaningful model of a program in certain modelling languages. Due to the limitations of some modelling languages it may not be possible to express the level of detail that is needed to verify properties of interest in a program.

Another common problem is the overhead associated with learning the syntax and semantics of any one of the many model checking languages that exist.

Lastly, even if an appropriate modelling language is chosen and an individual learns the modelling language, there is a problem with verifying that the model that is created accurately represents the original program.

2.2.2 State Explosion Problem

The State Explosion Problem is due to the exponential increase in the size of a finite-state model as the number of system components grows. This is more easily understood if one takes the example of a program written with only boolean/bit variables. Since each variable can be either true or false (one or zero) as each new variable is added to the model, the number of states of the program is doubled. This occurs because now all of the program's states that existed before the addition of the new variable still exist with the newly added variable set to one, plus all of their states when the new variable set to zero.

2.2.3 Requirement Specification Problem

Part of the overall functionality of a program consists of desirable and undesirable properties. These are known as the specifications of a program. They can be as

detailed as "operation foo will never return zero" to as abstract as "no deadlock". The problem is how to translate these sometimes abstract concepts into a set of usable specifications in the modelling language in use.

2.2.4 Output Interpretation Problem

Just as there are difficulties in determining the mapping from the programming language to the modelling language, there is also a problem in determining the mapping from the resulting model back to the implementation programming language.

2.3 Model Checking Tools

There are a variety of tools that are available and all of them address the inherent problems with model checking tools differently. All of the tools that are listed below are capable of representing concurrency.

2.3.1 SMV (Symbolic Model Verifier)

SMV (Symbolic Model Verification) is a formal verification tool that allows a user to construct a model of the finite state machine they wish to verify. Then the user can enter a specification for SMV in temporal logic. The specification is simply a collection of properties that indicate particular events that should or should never occur.

SMV's native language is comprised of simple data types such as bit, boolean, numerical variables and fixed arrays, though static structured data types can also be

constructed. As would be expected, the developer/verifier also has access to conditional constructs, such as case statements.

As the majority of the structure of SMV programs is comprised of simple data structures such as boolean and bit variables, an entire SMV program can be expressed as a finite state machine where each variable can have a value of one or zero (true or false). Thus since each variable has two possible states, if a single program has n variables, the entire program has 2^n possible states. As previously mentioned, this exponential growth in the number of states in a program is known as the *State Explosion Problem*.

To avoid having to traverse the entire state space and thus avoid the *State Explosion Problem* SMV uses Binary Decision Diagrams (BDDs). Essentially BDDs often give you a more compact representation of a FSM than the corresponding explicit, graph-like representation.

SMV addresses the *Requirement Specification Problem* by allowing the user to enter the specifications for their program in one of two types of temporal logic. LTL (Linear Temporal Logic) and/or CTL (Computation Tree Logic) can be used as specifications for a program depending on which flavour of SMV is being used (i.e. NuSMV, Cadence SMV or the original Carnegie Mellon University SMV). Both LTL and CTL are well suited to entering specifications. The essential element to both LTL and CTL is the use of primitive propositions. In the context of a program, a primitive proposition is a statement/expression which will evaluate to true or false.

LTL is able to express constraints on sequences of events or states or constraints on execution paths[Hat01]. Using LTL a user can make specifications:

- A proposition P is always true,

- P is eventually true eventually or
- P is true until another proposition Q is true

Here are the basic set of Universal and Existential quantifiers available in CTL:

- Along all paths proposition P holds globally
- There exists a path where P holds globally
- Along all paths P holds at some state in the future
- There exists a path where P holds at some state in the future

CTL allows a slightly more granular expression of specifications. However, this is not to say CTL is more expressive than LTL[Hat01]. LTL syntax is slightly easier to understand than CTL, however both temporal logics are capable of expressing an aspect of temporal logic that the other is incapable of expressing. For the most part, both temporal logics are capable of expressing the same concepts, but in a small number of instances, one is more expressive than the other[Sch96].

Due to the limited structural syntax in SMV (i.e. only static structures) it would be difficult to reliably complete a semantics preserving transformation on a program written in a third generation language such as Java into an SMV program. Though a limited transformation has been done from Java to a variety of modelling languages such as SMV and Spin[HT01]. This difficulty arises primarily because simulating the behaviour of complicated objects, such as those associated with I/O, requires more extensive knowledge of the underlying architecture.

Very little of either the *Model Construction Problem* or the *Output Interpretation Problem* is addressed by the creators of SMV. There has been some work in the

area performing an automated transform from a programming language to SMV in order to address the *Model Construction Problem*. However, there has been even less work in attempting to create a mapping from the original SMV program back to the original programming language. Since the inception of SMV, several translators have been developed that will perform SMV translations. These include:

- BIRC 0.5[IDH03] - Java to SMV, SPIN or dSPIN
- VeriTech[GK00] - SMV to Spin and back again

However these "translators" will not provide a method of mapping each line of the source language to each line(s) of SMV. Therefore, the problem of the *Output Interpretation Problem* still exists.

2.3.2 SPIN (Simple Promela Interpreter)

Spin is a tool that uses Promela (Protocol/Process Meta Language) to analyze data communication protocols of concurrent systems. There are two modes in which Spin can be used. The first is referred to as simulation; this is essentially attempting random simulations of the system's execution. The second mode called verification uses a set of correctness properties that Spin will verify. Spin is able to verify these properties by generating a C program that accurately represents the program that was entered in Promela. The verification mode can "verify the correctness of system invariants, it can find non-progress execution cycles, and it can verify correctness properties expressed in next-time free linear temporal logic formulae"[Hol97a].

In either mode, Spin will check for the absence of deadlocks, unspecified receptions, and unexecutable code.

The modeling language that Spin uses (Promela) consists of processes, channels and variables.

Typically, only a small subset of the overall functionality of a program should be implemented in Promela. This is done for two reasons, the first being that the transformation can be quite difficult to perform without some automated transformation tool, such as Bandera[CDH+00]. The second reason is that Spin completes an exhaustive verification in order to prove with mathematical certainty that a particular behaviour is error-free. Therefore as the size of the program grows, typically there is a complementary exponential growth in the length of time for Spin to execute (the *State Explosion Problem*). This is reduced somewhat by Spin's "bit storage technique", also known as supertrace[Hol97a].

Spin provides the user with the ability to enter specifications in the form of LTL. As previously mentioned LTL is an excellent manner to address the *Requirement Specification Problem*. Though few people are versed in LTL, it is a concise language that integrates temporal concepts.

Just as transformational applications have been developed for SMV to address the *Model Construction Problem*, there have also been transformational applications developed for Spin. These include:

- Bandera[CDH+00] - Java to Spin
- VeriTech[GK00] - SMV to Spin and back again
- Model checking SDL with Spin[BDHS00] - SDL to Spin

Again, as is the problem with SMV translators, Spin translators do not address the *Output Interpretation Problem* since they do not provide a user visible mapping

from the Promela code back to the originating language.

2.3.3 Rational Quality Architect Realtime Edition

RQA-RT (Rational Quality Architect Realtime Edition) is an add-in for Rational Rose RealTime program. Essentially Rational Rose RealTime allows developers to design, implement and test their software all in one tool. With the added functionality of RQA-RT a developer is also given the ability to verify the behaviour of their software.

Often during the development process multiple components are being developed in parallel. Often there are interdependencies between multiple components. In the situation where a component has a dependency on incomplete components, it is necessary to include stubs for those incomplete components.

RQA-RT allows stubs to be created which specify the expected behaviour of the system. The behaviour of each of the capsules in the system can be specified and the entire system can be run using the specifications. In this case, all tests will pass because the specifications for the system are the only thing being executed.

As each capsule is developed, each of them can be integrated into the implementation and verified against the behaviour which was specified in the initial specifications.

Some of the advantages of using RQA-RT are that:

- Each of Rational's models is capable of generating real code (C++, Java, etc) that functions as specified by the model

- Code Comprehension - As the majority of a program written in Rational Rose RealTime is comprised of UML (Unified Modelling Language) figures, it is much easier to understand the overall system by examining the structure of these

figures rather than trying to read line after line of the program's implementation programming language.

- Both the *Output Interpretation Problem* and the *Model Construction Problem* are addressed as the UML figures used in Rational are both the modelling language and the implementation (at least from an abstract view - real code is actually generated based on the UML implemented by the developer).

- *Requirement Specification Problem* - this is addressed by allowing the developer to create a very intuitive and user friendly sequence diagram. This sequence diagram will specify the expected behaviour to be verified. However, the verification capacity is somewhat limited in this regard in terms of the types of properties that can be verified. For instance, it isn't possible at present to enter properties/behaviors that are undesirable (for example: deadlock, livelock, etc).

2.3.4 Verisoft

Verisoft is unique in its ability to achieve complete coverage of the state space up to any desired depth. Of course the deeper the depth is set the longer Verisoft will potentially run.

Verisoft addresses the *State Explosion Problem* by making use of partial order reductions. The basic premise behind partial order reduction is that not all interleavings of concurrent events have to be examined. That is, the interleavings that correspond to the same concurrent execution in the state space need not be explored individually[God96].

Another way Verisoft reduces the state space is by only keeping track of visible operations. Visible operations are those operations which utilize Verisoft's C or C++

libraries. These include, but are not limited to, message passing operations and nondeterministic choice point operations.

VeriSoft is also capable of analyzing Java applications. Specifically, a whole Java virtual machine can be viewed only as a single black-box by VeriSoft. However, there is currently no way to monitor the executions and interactions of individual Java threads[God01].

Verisoft addresses the *Output Interpretation Problem* and the *Model Construction Problem* in a manner similar to Rational. As Bran Selic, Principal Engineer at Rational, is known to remark "the model is the implementation"[Sel03]. That is to say, the modelling language is C/C++. Therefore, a developer could write an entire program in C or C++ and then simply insert the appropriate Verisoft code to verify their code. The one disadvantage of using Verisoft is that it does not support dynamic process creation. In other words, one must know the number of processes to be created at compile-time.

Verisoft's ability to address the *Requirement Specification Problem* is limited to checking for divergences, livelocks, deadlocks or simple logical assertions which must evaluate to true or false using C/C++ syntax.

Verisoft can run in three different modes:

Manual Simulation Mode

In this mode, the user drives the simulation. This includes choosing which:

- process will execute the next line of code
- Value is selected at a nondeterministic choice point

This mode is useful if the user suspects where the failure (deadlock, livelock, etc) is occurring. Thus if there are a large number of states in the program, it may be quicker to step through the program to the point where the suspected failure is occurring.

Automatic Simulation Mode

In this mode, Verisoft will run without the need for the user to make any choices. The state space of the program is traversed in a breadth first search ensuring that, if there is a failure (for example: deadlock, livelock, etc), the shortest path to it is found first.

In the Automatic Simulation Mode execution the error trace, that led to the execution failure, is saved in a file named "error1.path".

Guided Simulation Mode

Automatic Simulation Mode must be run before Guided Simulation Mode can be run. After the "error1.path" file is created, this can be used to explicitly show the user the paths that were taken by each of the processes that lead to the error (i.e. deadlock, livelock or divergence).

2.4 Source Transformation

Source to source transformation provides developers with the ability to remove much of the dull and/or tedious work of recognizing simple and potentially complex patterns in code and updating them accordingly. By using a transformational programming language a developer is able to write a set of rules that can be applied to source code. These rules (provided they are valid/correct), when run on the original source code,

will produce syntactically valid transformed code. Essentially it allows the unique to write code that will itself write (transform) code.

2.4.1 Different Forms of Source Transformations

There are many different types of source to source transformations. Generally these different types of transformations fall into one of two categories[Vis01].

Translation

A Translation is a transformation from a language X into another language Y, where X is not the same as Y. That is to say, the source language and the target language are two different languages.

There are several different types of *Translation*. These include, but are not limited to, *Synthesis*, *Compilation*, *Migration*, *Reverse Engineering*, and *Analysis*.

Rephrasing

As implied by the name, a rephrasing involves a transformation within the same language but merely stated a different way. The different types of rephrasings are *Normalization*, *Optimization*, *Refactoring*, and *Renovation*.

2.4.2 TXL

TXL, was developed over ten years ago to be used as a tool for exploring programming language dialects. Since that time, TXL has been used for a variety of source transformations ranging from simple syntactical replacements to sophisticated software engineering transformations[CDMS02]. Further, TXL has been widely used in

research applications in industry and academia as well as in production commercial applications handling inputs of up to 100,000 source lines per input file[TXL02].

TXL is a pure functional programming language specifically designed to support structural source transformation. The structure of the source to be transformed is described using an unrestricted ambiguous context free grammar from which a parser is automatically derived. While based on a top-down approach, this parser has full backtracking and ordering heuristics to resolve both ambiguity and left recursion. The transformations are described by example, using a set of context-sensitive structural transformation rules from which an application strategy is automatically inferred. The rules are constrained to be homomorphic (type preserving) in order to guarantee a well-formed result.

2.5 Networking Basics

In order to network or connect two or more computers, there must be a physical and a logical connection. The physical connection is the wiring/cables that are used to connect two or more devices and over which data will be transmitted. The logical connection is a much more abstract idea. Basically the logical connection is achieved through non-physical ports.

These ports are essentially programming constructs that support the receipt and transmission of data. Certain applications communicate data over the aforementioned ports. For example, client applications, such as web browsers, and server applications, such as web servers, communicate using ports. Web browsers typically connect to port 80 of a web server.

Typically when a server program is started up it binds itself to a particular port

and waits for client programs to attempt to bind to the port being occupied by the server program[Wha01].

2.6 Client/Server Architectures

Client/Server architectures are network architectures that have two distinct roles. The server side of the architecture is a process that waits and listens for a client to connect to it. The client side of the architecture is the process that makes the connection to the server.

Connections are made via socket applications that are bound to specific ports[Sun03b]. Socket code is low-level application code that allows connections to be made on specified ports after the socket code binds to a specified port.

Peer to Peer applications are simply those applications where a process is both a client and a server.

2.7 Java RMI

2.7.1 Overview of Java RMI

The basis of Java RMI is very simple. The idea is for a remote object to "register" itself with the RMI registry. Registration involves giving the registry a unique name for the remote object being registered. In this sense, the registry acts like an internet accessible hashtable. Then after being registered, any Java object can query the registry for a reference to the remote object, using the unique name the remote object used to register itself. If an object queries the registry it gets a stub class object which

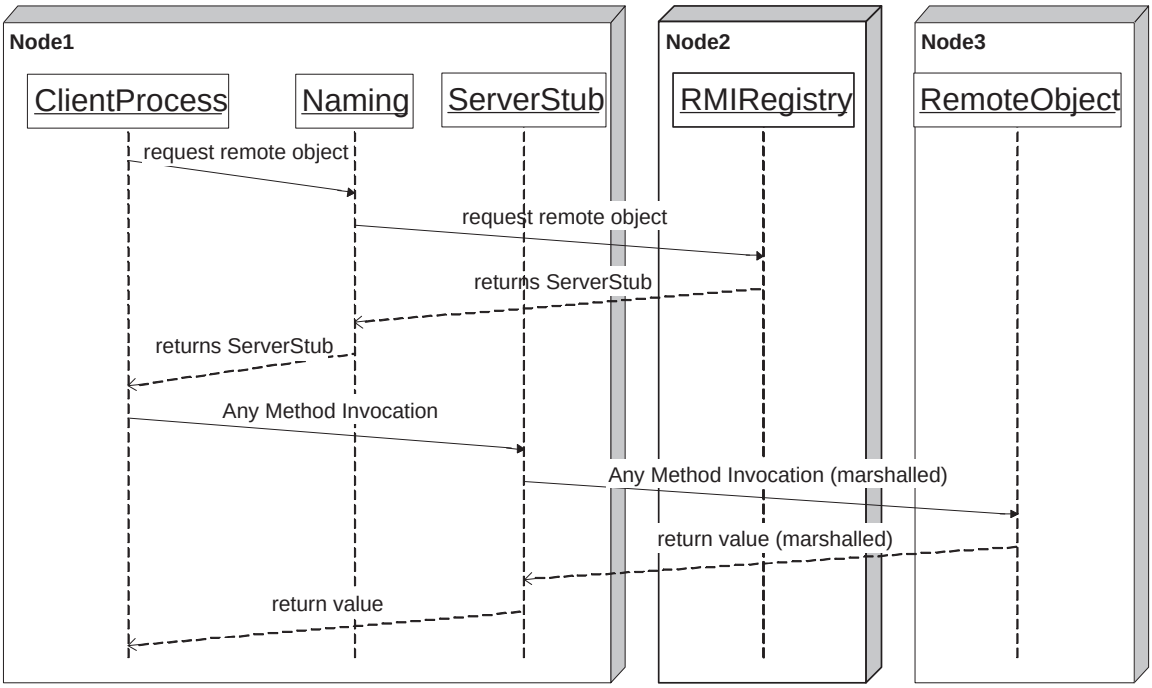

Figure 2.1: The sequence diagram for a Remote Method Invocation in Java RMI

has the same interface as the remote object. The stub class methods contain socket based requests to the remote object with the parameters marshalled and blocking code that awaits for the receipt of the return object from that remote method. In some cases, there is no object to be returned, i.e. methods with a void return type, in which case the local object will simply block until it receives an acknowledgement that the remote method has completed. Figure 2.1 illustrates this process.

2.7.2 Implementation Details

Java RMI is a portion of the Java SDK[Sun03a] (Software Development Kit) API (Application Programming Interface) that abstracts away from the details of the use of ports/sockets in communication across the network. More specifically:

> It is a mechanism that enables an object on one Java virtual machine to invoke methods on an object in another Java virtual machine. Any object that can be invoked this way must implement the Remote interface. When such an object is invoked, its arguments are "marshalled" and sent from the local virtual machine to the remote one, where the arguments are "unmarshalled." When the method terminates, the results are marshalled from the remote machine and sent to the caller's virtual machine. If the method invocation results in an exception being thrown, the exception is indicated to caller.[Sun02]

Since Java RMI provides the developer with the ability to communicate across the network, typical concurrency problems can arise, such as deadlocks and livelocks. In the context of Java RMI, a server is simply the class whose methods are being called remotely. That is to say, if there is an object A and an object B and A wishes to call a method on the object B which is located on a remote machine, then B is the server object.

Server Side

One of the requirements for a server process to be visible to a client object is that it must implement the java.rmi.Remote interface. However, in addition, any methods which are intended to be called by a remote object must be placed in an interface that extends the java.rmi.Remote interface. That interface must be implemented by the class whose methods will be called remotely. See Appendix A.1 for an exception to this rule.

In addition, each method that will be called remotely must fulfill the following[Sun01]:

- Must include the exception java.rmi.RemoteException (or one of its superclasses such as java.io.IOException or java.lang.Exception) in its throws clause, in addition to any application-specific exceptions (application-specific exceptions do not have to extend java.rmi.RemoteException).

- A remote object declared as a parameter or return value (either declared directly in the parameter list or embedded within a non-remote object in a parameter) must be declared as the remote interface and not the implementation class of that interface.

In addition, a server class is required to implement an interface that extends the java.rmi.Remote interface. The server class typically extends java.rmi.server.UnicastRemoteObject[2]. By extending the UnicastRemoteObject (in the java.rmi.server package) the class is given access to the remote behaviour of java.rmi.server.RemoteObject and java.rmi.server.RemoteServer[3].

It is also worthwhile to mention that making a server class with methods that can be invoked remotely doesn't mean that some of the methods cannot be invoked locally. That is to say it is legitimate to implement methods in a server class that have not been declared in the server's remote interface. However, these methods can only be invoked locally.

A server must also bind its unique name to the RMI registry. This allows clients to be able to "find" the server through the RMI registry.

[2]Note that if necessary, a class that implements a remote interface can extend some other class besides java.rmi.server.UnicastRemoteObject. However, the implementation class must then assume the responsibility for exporting the object (taken care of by the UnicastRemoteObject constructor) and for implementing (if needed) the correct remote semantics of the hashCode, equals, and toString methods inherited from the java.lang.Object class.[Sun01]

[3]This is beneficial because RemoteObject provides the remote semantics of Object and RemoteServer provides the framework to support a wide range of remote reference semantics. Specifically, the functions needed to create and export remote objects[Sun01].

Once the server code is completed, that code must be compiled with the RMI compiler. By doing this, the skeleton code for the server is generated[4]. The skeleton code handles all of the underlying networking needs of the communication. This includes, but is not limited, to setting up a connection, accepting the marshalled method invocation and potentially accompanying parameters and sending a response.

Client Side

A client can get a reference to the server by using the java.rmi.Naming class. The java.rmi.Naming class also provides access to services such as binding (already mentioned for the server process) unbind, lookup and listing the name-object pairings maintained on the host.

Upon completion of the client code, the code must be compiled with the RMI compiler, thus generating the client stub code. The client stub code is used to send the marshalled messages to the server process and to receive and unmarshall the response from the server.

2.8 Related Work

2.8.1 Limitations of Other Transformational Tools

Most transformations that are done by other tools miss aspects of the underlying semantics of the language[Hav99, CDH+00].

An example of this is the transformation done by Bandera[CDH+00] in transforming Java to Promela. In this transformation important dynamic I/O functionality,

[4]Skeleton code was made obsolete as of Java 1.2. The responsibilities that once belonged to the skeleton code now make more use of java.lang.reflect package.

such as sockets, files, etc, are impossible to transform. Though it is possible to model this functionality by indication of whether methods are blocking or non-blocking, dependency relationships, etc.

However, because C++ is a language that is capable of any I/O activities that Java is capable of, it will be possible to emulate the dynamic I/O behaviour of Java in the generated C++ program that uses Verisoft libraries.

Neither Bandera nor Java PathFinder[Hav99] (another transformational modelling tool) are capable of transforming Java RMI into a modelling language[Wal03, Sto03]. The main problem lies in the abundance of native methods in the Java RMI framework.

2.8.2 Advantages of Other Modelling Tools

Bandera's greatest advantage over Verisoft is its ability to allow the user to use Linear Temporal Logic or Computation Tree Logic to create the requirements specifications for a program. Basically, this means Bandera allows a much richer specification of what properties should or should not ever occur in a program. Similarly Java PathFinder is able to transform the Java code into Promela[Hav99]. Once transformed, the resultant Promela can be analyzed using Linear Temporal Logic.

These analysis tools are well suited to analyzing programs that make use of concurrency using modelling languages (such as SMV or Promela) that are more limited in their I/O capabilities than Java. In these instances, the behaviour of the various I/O libraries must be modelled. That is to say, the essential properties, such as blocking, nonblocking, dependency relationships, etc of the I/O functionality must be determined and then modeled/expressed in the particular modelling language.

However, there is a significant reduction in the state space of a program if the behaviour is modelled. Therefore, while these programs must model any dynamic I/O libraries it results in significant savings in the state space and thus a reduction in the time to determine properties of interest, such as deadlock, divergence, and livelock (results in the *Model Construction Problem*).

Therefore, while tools like Bandera and Java Pathfinder do suffer from the requirement of having to manually build models of most low level Java I/O libraries, the subsequent state space in their programs may be significantly reduced as a result.

2.9 Summary

There are many difficulties that arise in Software Engineering that can be addressed using model checking/verification tools. However, model checking/verification tools are not without their faults. They suffer from a variety of problems, not the least of which is the *Model Construction Problem*, that is constructing the model of the program and then validating that the model accurately represents the program.

The modelling/verification tools that were examined were SMV, SPIN, Rational Quality Architect Realtime Edition and Verisoft. Verisoft is unique amongst the tools in that it is able to directly analyze code written in two leading edge programming languages (C and C++). This essentially eliminates the *Model Construction Problem* that is often associated with modelling/verification tools.

Another area which had to be researched before my implementation could begin was the area of source transformation. There are many different types of transformation that can be achieved with transformational tools like TXL.

One other area of interest in my thesis was to find a specific implementation

for concurrency that could be analyzed. Since the infrastructure created for Java RMI allows developers to abstract away from the underlying networking details of distributed applications, it made it easier to model only the concurrency aspects of the program, as opposed to modelling the specifics of the networking code as well.

Lastly, it was necessary to examine existing tools that do transformations of a similar nature to the type that I am proposing. The two tools that seemed most significant in this area were Bandera and Java Pathfinder.

The next chapter will discuss the rationale for some of the high level choices that were made in my implementation. In addition, an overview will be presented to illustrate the steps involved in my implementation.

Chapter 3

Overview

As discussed in Chapter 2, the *Model Construction Problem* is one of the primary problems in using Model Checking tools. In short, the question is, how can a model be extracted and then constructed from the source code of a particular language? This sort of task is typically well suited to *transformational software* for several reasons.

3.1 Benefits of Using Transformational Software

Firstly, performing any transform by hand is prone to error. Even if both source and target languages are well understood by the person doing the transform, any transform of significant size poses a great number of opportunities for error. Conversely computer programs can easily be made to run in a deterministic manner. Therefore, their output is predictable.

Another reason for using transformational software is that a type of transformation can be proven to be correct (using formal proofs) for every instance of its application. In the case of doing the transformation manually it would be necessary to prove every

single change that was completed even if one change was of the same type as another.

Lastly, transformational software is used because of the speed with which its transformation of a large program can be performed. For large programs, not only would a human performing the transformation by hand be more prone to error, but it would take days to transform a file of 100,000 lines of source code.

Moreover, transformational software was used in this case as it does not require any changes to Verisoft. That is to say, I didn't have to expand the Verisoft libraries to allow for the analysis of Java applications.

3.2 Rationale For Choice of Verisoft

My main reason for using Verisoft for the analysis is that it works directly on source code. Being able to perform the analysis on the source code itself has two advantages. First, the possibility of spurious analysis results is reduced because no model *needs* to be constructed. Second, relating analysis output like error traces or counterexamples back to the source code is much easier.

Another benefit of Verisoft over other analysis tools is its partial order reduction. The basic premise behind partial order reduction is that not all interleavings of concurrent events have to be examined. That is, the interleavings that correspond to the same concurrent execution in the state space need not be explored individually[God96]. Consequently, partial order reduction has proved to be an effective means to keep the state explosion problem in check.

However, the message-passing modelling that Verisoft performs is inter-process communication, so any message passing that occurs in Java across the network will have to be reduced to inter-process communication.

3.3 Types of Transformation Used

As discussed previously in the background, the two major categories of transformation are *translation* and *rephrasing*. A *translation* is a transform where source and target languages differ. A *rephrasing* is a transform where the source language and the target language are the same. The transformation that will be used is both a translation and a rephrasing. The first step of my implementation will be a translation and the second step will be a rephrasing.

The type of translation that will be used is a migration. A migration is when a program of one language is transformed to another language while maintaining the same level of abstraction[Vis01]. In this instance, the change will be from the Java source language to C++ source code. But since the change will not be a semantically preserving transform in regards to messages that being passed over the internet (i.e. using ports/sockets) there will also need to be a rephrasing of the code.

The rephrasing that will be done is a type of renovation. A renovation is simply a change to software as a result of a change in the software requirements.

3.4 Outline of Solution

Figure 3.1 shows the overall structure of my transformation. There are three basic steps to go from a valid Java RMI program to a program being analyzed by Verisoft. The first step (Java to C++) is an automated transform from Java to C++ using TXL. The resulting C++ can then be compiled and executed providing that any and all of the Java libraries that the original Java code depended on have also been transformed.

The second step requires the generation of a class that models the functionality of both Java Naming and the RMI registry. This step also involves the generation of stub classes, which act as proxies for remote objects, and the UnicastRemoteObject class. All of which is performed with programs written in TXL.

The third step is compiling, linking and then executing the resulting C++ code in Verisoft. Verisoft can then be used to analyze the resulting model.

3.5 Summary

Since programs that support concurrent execution of code are difficult to debug, external tools are sometimes used to determine the source of problems. However, modelling/verification tools are often written in difficult to understand languages. Verisoft is a concurrency analysis tool that does not require a model to be built, since it makes use of C and C++. However, many programs are written in languages other than C and C++. Manual transformations from a source language to a modelling language can be problematic. However, automated source transformation is less error prone and more scalable than completely manual transformations and can therefore be used to produce valid models of a program.

This implementation makes use of the two major forms of source transformation, including a translation and a rephrasing. The first step will make use of a translation and the second step will make use of a rephrasing.

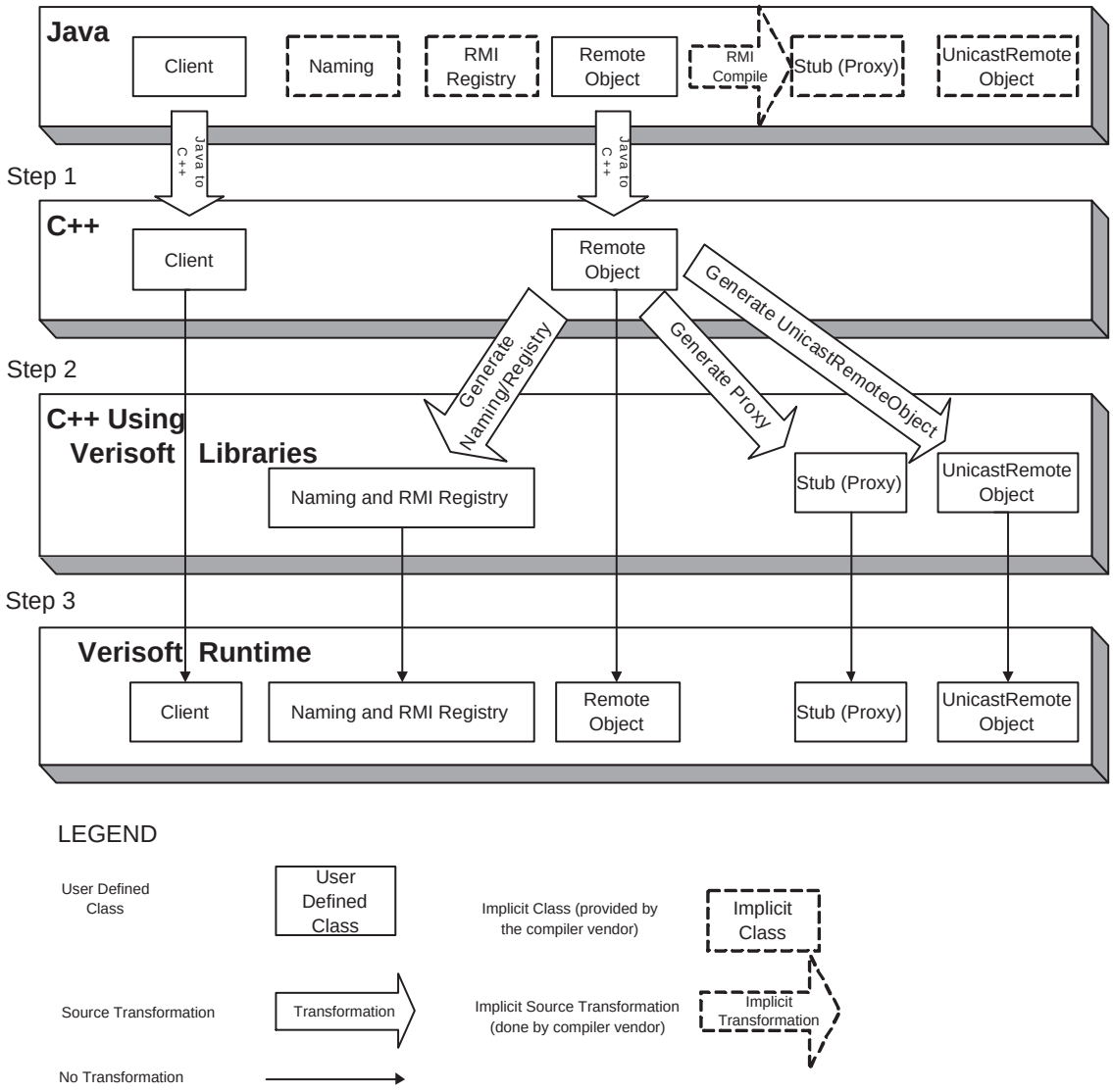

Figure 3.1: The sequence of transformations and subsequent execution of the model

Chapter 4

Step One:

Java to C++

The first step in the process is a semantics preserving transformation from Java to C++ (see Figure 3.1). There are a variety of considerations that had to be made in performing this transformation. These include, but are not limited to, keywords in Java that do not have counterparts in C++, constructor mechanisms, and the usage of arrays. In automating this transform, a number of limitations arose, which will also be discussed.

4.1 Reduction in Requirement for Full Semantic Transformation

Since this transformation is a semantics preserving transformation and intended only for model checking of the program it is not necessary to ensure that compile-time

accessor restrictions, such as protected and private, are maintained. That is to say, the program that is transformed must already have been compiled and functionally tested before beginning the transformation from Java to C++. Because any compile-time restrictions will have been assured by the Java compiler before the transformation is performed, JCUV need not enforce any compile-time restrictions.

However, in all cases where there was a requirement for compile-time restrictions, a best effort transformation was done to maintain those restrictions in C++. This was done to support future work that might make use of this transformation software to complete a fully automated transform from Java to C++.

4.2 First Step Transform Examples

To show the second step transform in action, two sets of example code have been transformed and explained below.

4.2.1 Interesting Aspect of Java Transformed

One example of the types of transformation being done at this stage is the transformation of arrays. Though arrays are a simple construct, Java's arrays actually extend java.lang.Object. Therefore, the following is legitimate Java code:

```
1 int [] arrayOfInts;
2 Object javaObject = arrayOfInts;
3 int [] newArrayOfInts = (int []) javaObject;
```

Thus there was a requirement that the transformed java.lang.Object class and arrays (in C++) support this sort of assignment. Therefore a class was written which

essentially acts as a wrapper class[1] around C++'s standard vector class and which also extends the transformed java.lang.Object class. The name of this new class was `StdVector` and thus the transformation had to transform any Java array into an object of type `StdVector` in C++.

The above Java code example of array assignment can thus be automatically transformed into:

```
1 StdVector<int>::type arrayOfInts;
2 SmtObjectPtr javaObject = arrayOfInts;
3 StdVector<int>::type newArrayOfInts = javaObject.Dynamic_cast((
    StdVector<int>::type) 0);
```

One of the transformational rules for Java array declarations to C++ `StdVector` declarations is illustrated in Figure 4.1. The rule is being used to replace a Java variable declaration with a C++ variable declaration. The pattern that is being sought is one in which there is (in this order):

- An access specifier - public, private, etc
- A series of modifiers - static, final, etc
- A type name - String, Hashtable, int, etc
- A declared name - any series of alpha-numerical characters that aren't reserved words
- A subscript potentially containing an expression - for example: a+b
- A semi-colon

The replacement is (in the following order):

[1] A wrapper class is a class that allows access to the services of another class through its own methods.

```
rule arrayDeclarationAndArrayDefinitionTransformation
 replace [variable_declaration]
   Mods [repeat modifier] TypeName [type_name] '[ OptExpression [opt expression] ']
   VarName [variable_name] '= 'new AssignedTypeName [type_name] '[ SizeOfArray [opt expression] '] ';
 by
   Mods 'StdVector '< TypeName '> ':: 'type VarName (SizeOfArray) ';
end rule
```

Figure 4.1: A transformation from Java arrays to objects of type `StdVector`

- The same access specifier - public, private, etc
- The same series of modifiers - static, final, etc
- The type name - StdVector with a template parameter that contains the type name from the original statement followed by two colons and the word "type"
- The declared name from the original statement
- A parameter to the constructor that is the size of the array from the original statement
- The semi-colon

4.2.2 Deadlock Code From Figure 1.1 Transformed

Figure 4.2 is the RMI deadlock code from Figure 1.1 after it has been transformed using the second step transform.

4.3 Memory Management

Memory management is a necessity in any program since all computers have a finite amount of memory. Programming languages handle memory management in different

```
#ifndef PeerA_H
#define PeerA_H
class PeerA;
typedef SmartPtr <PeerA> SmtPeerAPtr;
class PeerA : public UnicastRemoteObject,
              public PeerAInterface,
              public Serializable {

  //removed addRef and release methods
  //for readability

  public:
    virtual void callBack () {
        Synchronized dataGuard(*this);
        //never make it into here
    }

  public:
    virtual void run () {
        Synchronized dataGuard(*this);
        try {
            SmtStringPtr name = "PeerB";
            SmtPeerBInterfacePtr peerB =
              (Naming->lookup(name)).Dynamic_cast(
                (SmtPeerBInterfacePtr *) 0);
            peerB->executeTask();
        }
        catch (SmtExceptionPtr exception_) {
            exception_->printStackTrace();
        }
    }

};

int main (int argc, char * args []) {
    try {
        SmtStringPtr name = "PeerA";
        SmtPeerAPtr peerA (SmtPeerAPtr(
          new PeerA ()));
        Naming->rebind(name, peerA);
        peerA->run();
    }
    catch (SmtExceptionPtr exception_) {
        exception_->printStackTrace();
    }
}

#endif
```

```
#ifndef PeerB_H
#define PeerB_H
class PeerB;
typedef SmartPtr <PeerB> SmtPeerBPtr;
class PeerB : public UnicastRemoteObject,
              public PeerBInterface,
              public Serializable {

  //removed addRef and release methods
  //for readability

  public:
    virtual void executeTask () {
        try {
          SmtStringPtr name = "PeerA";
          SmtPeerAInterfacePtr peerA =
            (Naming->lookup(name)).Dynamic_cast(
              (SmtPeerAInterfacePtr *) 0);
            peerA->callBack();
        }
        catch (SmtExceptionPtr exception_) {
            exception_->printStackTrace();
        }
    }
};

int main (int argc, char * args []) {
  SmtStringPtr name = "PeerB";
  try {
    SmtPeerBInterfacePtr peerB (
    SmtPeerBPtr (new PeerB ()));
    Naming->rebind(name, engine);
  }
  catch (SmtExceptionPtr exception_) {
    exception_->printStackTrace();
  }
}

#endif
```

Figure 4.2: Simple RMI Java code example transformed into C++ (reference counting overhead code has been elided from this example to make it more readable).

ways.

4.3.1 Memory Management in Java

Java uses an advanced garbage collection algorithm that runs in the background while a Java program executes. Therefore, memory management is trivial for any Java developer. C++, however, does not inherently possess a garbage collection algorithm like that which is found in most Java Virtual Machines.

To bridge this gap, my approach was to write a rudimentary garbage collection algorithm that involves smart pointers and reference counting.

4.3.2 Memory Management in C++

The stack is used to manage automatic objects in the program. Whenever entering a function or a block of code (code surrounded by "{" and "}") automatic objects are created automatically. They are also destroyed automatically whenever the flow of control in the program exits the function or block of code[Kal99].

Whenever the keyword **new** is used in C++ to create an object, it is created on the heap. Since, this memory is not automatically reclaimed, it is necessary to explicitly use the **delete** keyword to reclaim that memory, otherwise a memory leak is created[LAH+99].

The concept of ensuring that every call to **new** is matched by a **delete** sounds obvious and easy to implement, but often it is difficult to determine when a section of code is "finished" with an object. One of the primary difficulties is when exceptions are thrown.

Here is a section of C++ code to illustrate the problem concerning freeing memory

using the **delete** keyword:

```
1  int main (){
2    try
3    {
4      TrainCar * test1 = new TrainCar (100);
5      TrainCar * test2 = new TrainCar (10);
6      foo (test1);
7      delete test1;
8      delete test2;
9    }
10   catch ( myException & e )
11   {
12     cout << "caught exception: " << e.errorMsg() << "\n\n" <<
           endl;
13   }
14   return 0;
15 }
```

In the above section of code there is no explicit declaration of either the method `foo` or the exception class `myException`. As we will see later, just what the function `foo` is doing becomes critical. However, without even considering whether the `foo` function uses the **new** or **delete** keywords, there are already some problems with freeing memory in the example above.

The first primary problem relates to the possibility of an exception being thrown. If an exception, of type myException, is thrown by the `foo` function then the flow of control will move from line 6 to line 10, execute the contents of the exception block and then exit the program. However, in this case, neither of the destructors for the classes have been called and the program has exited. This results in memory which cannot be reclaimed (the memory used by the two `TrainCar` objects) until the computer's memory is cleared (i.e. following a clean boot).

The second problem occurs if the destructor throws an exception (of type myException) on line 7. This means that not only will the destructor of the object `test2` not be called, but this also means that the destructor of the object `test1` did not

complete successfully. Therefore, the flow of control will move from line 7 to line 10 and subsequently the memory of at least one object will not be freed (memory allocated for object `test2`).

Finally, if the function `foo`, frees the memory (i.e. the **delete** keyword is used) and then we attempt use delete again on that pointer, a critical error can result.

To be more clear, when the **delete** keyword is used on a pointer the memory that the pointer is pointing at is freed, which thus allows new data (potentially other objects) to be placed in that section of heap memory. Once this is done, where the pointer will point is undetermined (differs between C++ compilers). Therefore, if a second **delete** statement is issued, it is undetermined which section of memory will be deleted which will thus result in undefined behaviour.

In conclusion, memory management in C++ is error prone and in some cases can result in critical errors. Therefore, the use of a garbage collection algorithm in many cases isn't just useful, but is often necessary simply to create tractable code. The next section describes my approach to supplying a rudimentary garbage collection algorithm.

4.3.3 Two Garbage Collection Classes

Since there are a number of problems with memory management in C++ that do not exist in Java two new classes will be introduced. The first class is a smart pointer class that makes use of templates. This class will allow dynamically created C++ objects to keep the advantages of sitting on the stack (automatic objects). Essentially this means that when the object "goes out of scope" and there are no more references to the object it will be removed from memory. This class essentially acts as a wrapper

class for the class of interest.

To allow for this self deletion and tracking the number of references that are made to an object, another class will have to be created that all other classes will extend[2]. It is the base class of all other classes. The only memory management tasks this class will have is to keep track of the number of references there are to it and then delete itself once there are no more references to it.

Smart Pointer Class Implementation

The Smart Pointer class is provided so that your classes have all the benefits of being on the stack (for example: will remove themselves from memory when they go out of scope) and all the benefits of being dynamically created (for example: object creation can be determined at run-time). This class is used to tell the base class when a new reference has been gained or an existing reference has been lost.

Base Class Implementation

The base class (in the C++ code written to support the transformation), named JCUVobject (the prefix JCUV is an acronym derived from Java to C++ Using Verisoft), is a class that all other classes will extend (much like the java.lang.Object class in Java). The sole purpose of the base class is to allow any class that extends it to delete itself when there are no more references to it.

In order to properly transform Java classes to C++ classes, it was necessary to provide the same hierarchy in C++ as exists in Java. All Java classes extend java.lang.Object directly or indirectly. Unfortunately, no Java class must explicitly

[2]Conveniently Java also has a base class that all other classes extend either directly or indirectly named Object (contained in the java.lang package).

extend java.lang.Object. In Java, if a class does not extend any class explicitly then it extends java.lang.Object implicitly. However, if a class does extend another class (other than java.lang.Object) it will only extend java.lang.Object indirectly through its superclass. Therefore, the transformed C++ java.lang.Object class extends the JCUVobject class to ensure that all classes extend JCUVobject and thus allow for the self deletion used in the JCUVobject class.

Using Smart Pointers in Transformed Code

In the transformation if there is a Java class `TrainCar` and there is an expression:

```
1 TrainCar trainCar = new TrainCar(42);
```

The transformation would make use of the smart pointer class (`SmartPtr`) and the base class (JCUVobject) already implemented in C++ code to facilitate this transformation. Thus, the transformed line of code in C++ would look like this:

```
1 SmartTrainCarPtr trainCar(new TrainCar(42));
```

In the above line, the `TrainCar` class would have been transformed over to C++ such that it extends the base class (JCUVobject) and the following typedef is used to enhance readability:

```
1 typedef SmartPtr<TrainCar> SmartTrainCarPtr;
```

4.4 Packages and Namespaces

Java **package** structures are analogous to C++ **namespace** structures. Java **package** and C++ **namespace** structures allow a developer to create libraries (sets of classes with a common name prefix). This is useful for two reasons:

1. It solves the name resolution problem that has plagued many programming languages. That is to say, the problem that arises when there are name conflicts between classes. For example, a list structure in a GUI package (with the class name List) and a list structure that forms the basis for Queues and Stacks (also with the class name List). This allows the two classes to be placed in their own unique package or namespace.

2. It provides more provision of structure (and therefore typically understandability) in a program or API (Application Programming Interface). Since there are logically multiple components to any single program or API (for example I/O, GUI, generalized data structures, etc - each of which can be further subdivided depending on the size of the API or program) there should be a way to partition related sets of classes up in the programming language itself.

The syntax of declaring a **package** or **namespace** differs slightly between Java and C++. For Java, the package for a class is put at the top of the file in which the class occurs. To make use of packages in other packages with the package qualifier an **import** statement must be used. For example:

```
1  package TestPackage;
2
3  import java.util.Vector;
4  import java.io.*;
5
6  //PackageExample is a class within the package TestPackage
7  class PackageExample {
8    //the File class is found in the java.io package
9    File m_file;
10   //the Vector class is specifically imported from the java.util
       package
11   Vector m_vector;
12   //the Hashtable class is being fully qualified
13   //so that neither its package nor it need to specifically be
       imported
14   java.util.Hashtable m_hashtable;
```

```
15 }
```

In C++, the classes to be included in a particular namespace must be delimited by brackets. In order to use classes in different namespaces, the developer can either make use of the **using** keyword or prefix the class/struct with the namespace followed by two colons. For example, the transformed version of the above Java code is (reference counting overhead code has been elided from the example below to make it more readable):

```
1  using namespace java::util;
2  using namespace java::io;
3
4  #include "Vector.h"
5  //The two lines below had to be manually added
6  #include "File.h"
7  #include "Hashtable.h"
8
9  //PackageExample is a class within the package TestPackage
10 namespace TestPackage {
11     class PackageExample;
12     typedef SmartPtr <PackageExample> SmtPackageExamplePtr;
13     class PackageExample : public Object {
14
15       //the File class is found in the java.io package
16       public:
17         SmtFilePtr m_file;
18
19       //the Vector class is specifically imported from the java.
             util package
20       public:
21         SmtVectorPtr m_vector;
22
23       //the Hashtable class is being fully qualified
24       //so that neither its package nor it need to specifically
             be imported
25       public:
26         java::util::SmtHashtablePtr m_hashtable;
27     };
28
29 }
```

In the above example, the lines:

`#include`"File.h" `#include`"Hashtable.h"

had to be added since Java allows an entire package of classes to be imported into a file. Whereas, C++ requires each individual file to be included using separate include statements. Therefore, anytime that a Java program imports a package of classes instead of importing a specific class, the equivalent C++ must explicitly import each individual file that contains that class.

4.5 Class to File Relationship

In Java, all information concerning a particular class must be defined within the class structure occurring in the Java file associated with that class[3]. However in C++ the definition and implementation of a class can occur in any file that is chosen by the developer.

However, C++ allows there to be an arbitrary number of classes in a single file. Therefore, since C++ is more flexible than Java in this regard C++ will allow me to create whatever file to class relationship I require.

4.6 Entry Method/Function

In both Java and C++, the entry method/function[4] is named "main". However, in Java, the main method is contained within a class whereas in C++ it is a global function belonging to no class. In Java, the user must explicitly invoke the class that contains the "main" method. In C++, so long as there was a global main function

[3]Multiple classes can be defined within a single Java file either by not declaring the other classes public or by using inner classes

[4]In this context entry method/function simply means the method or function that is the starting point for a program when that program is executed

somewhere in one of the source files (that is compiled into an object file and then linked), it will be found and executed in the respective executable code.

The method header for the "main" method in Java is:

```
public static main (String [] args)
```

According to the ISO/IEC 14882:1998 C++ standard [Int98], there are two versions of the definition of the main function in C++ that are acceptable:

```
1 int main() { /* ... */ }
```

and

```
1 int main(int argc, char* argv[]) { /* ... */ }
```

> In the latter form argc shall be the number of arguments passed to the program from the environment in which the program is run.[Int98]

One problem that emerges is that multiple Java class files within a program can have main methods, whereas in C++ only one global function can be named "main" (with the aforementioned properties). Therefore, JCUV assumes that all Java code that forms a program will contain only one main method within a class with the aforementioned properties. If this isn't the case, then the subsequent C++ code will have multiple main entry methods and will thus cause linker (if the duplicate main function occurs in a different file) or compiler (if the duplicate main function occurs in the same file) errors.

4.7 Unused Keywords

4.7.1 transient Keyword

Java has a keyword called **transient** which is used to mark member variables of classes that should not be saved during object serialization. Object serialization is the process of creating a byte array for easy transfer or storage of the object. This allows an object to be saved into the byte array and recreated at a later time and potentially on a different machine such that its state is maintained.

When a member variable is marked with **transient** it will not be saved into the byte array and thus will not be recreated when the object is "deserialized". However, since this is basically an optimization feature[5] (i.e. to reduce the size of serialized objects) it is not necessary for the transformed code to maintain this functionality.

4.7.2 abstract Classes, Variables, and Methods

In Java, a class that is intended to be **abstract** is denoted by putting the **abstract** keyword before the **class** keyword. Then any methods that are abstract have to be marked explicitly with the **abstract** keyword.

In C++, a class that is intended to be abstract need only have one method that is marked with the "=0" notation following the method declaration to be deemed abstract. In other words, the declaration of an abstract class in C++ is done implicitly by having at least one abstract method.

However, since Java is capable of having abstract classes that contain no abstract

[5]It is possible that a transient member variable could act as a marker to the developer as to whether the current object had been serialized or not. Also in some cases it may be that the information is not desired between sessions (if the serialization is being used to write data between sessions).

methods, it will be necessary to make classes that were abstract in Java regular classes in C++. That is to say, a Java class that is marked **abstract** will only be made into an abstract class in C++ if the Java class has at least one abstract method. Thus, the solution is to remove the `abstract` prefix from Java classes in the transformation. Then to replace the Java syntax of declaring an abstract method with the C++ syntax of declaring an abstract method.

4.7.3 The native Keyword

The **native** keyword is applied to methods to indicate that the implementation for the method will be provided in a language other than Java. In these cases, the native method's implementation is written manually such that the functionality of the C++ code matches that of the original Java method. Coincidentally, typically when the **native** keyword is used, the language that is typically used is either C or C++[6].

4.7.4 The synchronized Keyword

The **synchronized** keyword in Java acts like a monitor on blocks of code (including methods). If a block of code is marked with synchronized, only one thread at a time can enter that particular block of code for a particular instance of an object. This functionality is emulated by replacing Java code of this form:

```
1 class SynchronizedCode {
2   synchronized void someSynchroMethod (){
3     System.out.println ("You can't be told what the method is -
          you have to experience it - one at a time.");
4   }
5
```

[6]Since the transformation being completed is one from Java to C++ the first step of transformation may present Java code optimization possibilities by using the resulting generated C++ in Java native methods.

```
}
```

with C++ code of this form:

```
#ifndef SynchronizedCode_H
#define SynchronizedCode_H
class SynchronizedCode;
typedef SmartPtr <SynchronizedCode> SmtSynchronizedCodePtr;
class SynchronizedCode : public Object {

//removed AddRef and Release methods
//for readability

  public:
    virtual void someSynchroMethod () {
        Synchronized dataGuard(*this);
        System::out->println("You can't be told what the method
           is - you have to experience it - one at a time.");
    }

};

#endif
```

As previously mentioned, synchronization in Java can be specified on any block of code. For example:

```
class SynchronizedCode {
  void someMethodWithSynchronizationInIt (){
    synchronized (this){
      System.out.println ("You can't be told what the block of
          code is - you have to experience it - one at a time.");
    }
  }
}
```

Where the keyword `this` could be any Java expression.

4.7.5 The final Keyword

The **final** keyword in Java has the same syntactic meaning as the **const** keyword in C++. The **final** keyword marks a variable as being unchangeable. That is to say, once a value has been assigned to a final variable, that value can not be changed.

In addition, in Java there is a concept of denying permission to extend a class and/or override a method. This can be done by prefixing the **class** keyword by the **final** keyword, this indicates that this class cannot be extended.

In C++, a class can prevent other classes from extending it, by making its default constructor **private**. However, this isn't very useful since it creates another restriction on the class that did not exist in the original Java version. That is, if the original final class in Java had a public default constructor, the C++ version would have a private version.

In C++, there is no mechanism to stop a derived class from overriding another class's public methods. However, a derived class can be stopped from overriding a method polymorphically by removing the **virtual** keyword from a member function.

However, in the context of this thesis, there is no requirement to enforce compile-time restrictions as the Java code submitted to this program will already have been compiled.

4.7.6 The null Keyword

The **null** keyword in Java is identical in function to the **NULL** (or '\0') keyword in C++. Thus anywhere there is an occurrence of **NULL** in Java it can be replaced by '\0' in C++. Often it is useful to set references/pointers in C++ and Java to point at null. This technique is used to determine if an object has been initialized or possibly to ensure that a reference no longer references an object that it was previously referencing.

4.7.7 The volatile Keyword

The **volatile** keyword in Java is used to prevent the compiler from performing certain optimizations on a member of a class (i.e. methods and variables). For further information see Appendix B.

For the time being, the keyword will simply be removed until threading support is added to my framework. The reason for this is simply because the **volatile** keyword has no application in environments without thread support.

4.8 Constructors

4.8.1 Initializing Variables

In C++ it is legitimate syntax to initialize variables in a class in three different manners. One way is only supported for a member variable that is both const and static and simply involves assigning the member variable a value on the same line in which it is declared. Another way is simply to initialize the variables by way of the constructor of a class. For example:

```
1  class TrainCar : public Car{
2    int m_id;
3    public :
4      TrainCar (int idValue_):m_id(idValue_) {}
5  };
6
7  int main (){
8    TrainCar test1(100);
9
10   return 0;
11 }
```

In the above example, the member variable `m_id` is being assigned the parameter being passed into the constructor for `TrainCar`. This is a more concise way of

initializing member variables in classes than writing the statement:

`m_id = idValue_;` within the constructor's body (which technically is not "initializing" the variable - since the member variable will not start with that value from the moment of creation).

Java only has access to the first and last manners of initializing member variables (by assignment). Also in the first case, the member variables don't have to be static and final; the member variables can have any modifiers applied to them.

It should be noted that if a member variable in a Java class is not initialized explicitly then it is implicitly initialized with 0 (if it has a numerical type) or null.

However, in C++, those member variables are undefined initially. Therefore, it is necessary to ensure that any member variable that isn't explicitly initialized in C++ is initialized to null or zero, depending on its type.

4.8.2 Calling Superclass

In Java, the superclass constructor of a class is invoked by explicitly writing the expression:

```
super ();
```

The superclass constructor invocation must be the first expression encountered in the subclass's constructor. This call to the superclass's constructor is capable of accepting parameters just like any other class constructor. Here is an example of a base class constructor being invoked by a derived class constructor in Java:

```
1 class Car {
2   private int m_foo;
3   public Car (){
4     m_foo = 5;
5   }
6 }
7
```

```
8  class TrainCar extends Car{
9    private int m_id;
10   public TrainCar (int idValue_){
11     //Car's default constructor is invoked
12     super ();
13     m_id = idValue_;
14   }
15
16   public static void main (String [] args_){
17     TrainCar test1 = new TrainCar (100);
18   }
19 }
```

In C++, the syntax for invoking the constructor of a superclass is identical to the syntax for initializing a member variable of the class. For example the transform of the above results in (reference counting overhead code has been elided from the example below to make it more readable):

```
1  class Car;
2  class TrainCar;
3  typedef SmartPtr <Car> SmtCarPtr;
4  typedef SmartPtr <TrainCar> SmtTrainCarPtr;
5  class Car : public Object {
6
7    private:
8      int m_foo;
9
10   public:
11     Car () {
12         m_foo = 5;
13     }
14 };
15
16 class TrainCar : public Car {
17
18   private:
19     int m_id;
20
21   public:
22     TrainCar (int idValue_) : Car(){
23         //Car's default constructor is invoked
24         m_id = idValue_;
25     }
26
27 };
28
29 int main (int argc, char * args_ []) {
```

```
30      SmtTrainCarPtr test1 (new TrainCar (100));
31  }
```

In the above example TrainCar calls its superclass constructor by the following:

`Car()`

The remainder of the constructor simply initializes the value of its member variable id to the value of the parameter `idValue_`.

4.9 Using Arrays

In Java, the syntax for declaring an array is either

```
<type of variables in array> [ ] <name of array variable>;
```

or

```
<type of variables in array> <name of array variable> [ ] ;
```

The definition of the size of the array can be given on the same line as the declaration or at any point later in the program.

In C++, the syntax to declare an array is slightly different. If the developer knows the number of elements in the array at compile time, then the following format can be used:

```
<type of variables in array> <name of array variable>
      [<number of elements in array>];
```

If however, the program can only determine the size of the array at run-time, then the following format is used:

```
<type of variables in array> *<name of array variable>;
```

Then at some point later in the C++ program (or even on the same line as the declaration above) the array variable can be assigned the number of objects it will

point at. For example:

```
<name of array variable> =

        new <type of variables in array>[<number of elements in array>];
```

The main difference between the two methods in C++ is in the first case, the program will automatically construct the number of objects (automatic objects) that are desired when it reaches that line of code (and subsequently free that memory when they exit the current code block). In the second case, whenever the definition of the number of elements in the array occurs, the program will instantiate the number of objects that are specified. Those objects are instantiated by invoking the default constructor for each of those objects, but it will not automatically free that memory.

For example:

```
1  //an array of 10 pointers to the primitive data type "int" are
       created
2  //In addition the default constructors will be invoked on these
       Test objects
3  //To be more clear if the default constructor is made private -
       the following line
4  //will not compile
5
6  int *arr1 = new int[10];
7  //If any of the 10 pointers point to valid "int" objects they
       will be removed from memory
8  //(from the heap) when the delete statement is reached
9  delete[] arr1;
10
11 //an array of 10 "int" objects are created in memory
12 int arr2[10];
13 // and will be destroyed when out of scope
```

As is evidenced above, the manner of declaration of arrays differs between C++ and Java. Another point of difference between Java and C++ arrays is that Java arrays have a member variable `length` that indicates the number of elements in the array.

All of this created the impetus to write a container "array" class for C++ so that

a member variable `length` could be used to emulate this functionality.

There are methods for determining the length of a dynamically created array in C++; however, these methods can result in out-of-range errors or even more serious undefined behaviour. Therefore, it is much safer to use a "container" class such as the following classes that are found in the standard namespace: vector, list or map[7].

Therefore, in this implementation the vector (in the standard namespace) class will be used so that:

- The size of the "array" type class can be set after the "array" type class is declared. A regular static array wrapper class would not allow this (see Appendix A.4).
- The length of the "array" type class can be determined without the possibility of introducing problems such as "buffer overflow". Determination of the number of elements in the structure can be done using a member variable named length (as in Java).
- Subscripting is supported just as is it is for regular Java arrays. Subscripting is not actually supported by the list class in the standard as it is a sequence optimized for insertion and deletion of elements which results in the list class being much slower[Str97][8].

In addition, to customize the use of the vector class (in the standard namespace), I have provided a wrapper class called StdVector. The declaration of this class can

[7]The standard namespace (named std) was defined by the C++ ISO/ANSI standard as approved in 1998[Int98]. It contains a variety of classes to facilitate writing "standard" C++ code (i.e. data structures, input/output libraries, etc).

[8]An example of subscripting is:
intArray[0]
which accesses the first element in the array called intArray (assuming 0 indexing is used)

be found in Appendix A.5.

4.10 Polymorphism

A necessary characteristic of any system that supports polymorphism is dynamic binding. Dynamic binding means that requests of one type can be matched to multiple implementations that are resolved at run-time.

Therefore, one of the benefits of possessing a system with dynamic binding is polymorphism (though polymorphism is not always beneficial to a program). Polymorphism allows objects with identical interfaces to be substituted for each other at run-time[GHJV95]. A way of explicitly assigning an object of one type to an object of another type is through casting. In C++, the costs associated with having a system make use of polymorphism typically takes the form of having to search through v-tables (virtual tables) to resolve functions to classes at run-time. Thankfully both C++ and Java provide semantically equivalent polymorphic support, thus enabling the transform.

4.10.1 Casting

Casting objects from one type to another is a useful mechanism to take advantage of polymorphism. However, in order to use casting; the object being cast must belong to the hierarchy of the class that one wishes to cast to. That is to say, if there is an object of type A and an object of type B, in order to cast A to B or vice versa, the two must belong to the same class hierarchy.

The two types of casting are upcasting and downcasting. Upcasting is typically

implied and does not require an explicit cast in most languages.

Java provides flexible casting, in that it allows a user to either upcast or downcast explicitly. However, if an object is being downcast, explicit casting must be used. For example, in Java:

```
1  class Test{
2
3    public static void main (java.lang.String [] _args){
4      java.lang.Object testObject;
5      //java.lang.String extends java.lang.Object
6      java.lang.String testString = new java.lang.String ("testing"
           );
7      //testObject gets a reference to the object that testString
           references
8      testObject = testString;
9      //now testString2 gets a reference to the object that
           testString references
10     java.lang.String testString2 = (java.lang.String) testObject;
11   }
12
13 }
```

On line 8 there is an example of implicit casting and on line 10 there is an example of explicit casting. For instance, the casting that occurs in Java can be done in the resulting C++ in the following manner (some C++ memory management code was removed from the example below to make it more readable):

```
1  int main () {
2    java::lang::SmtObjectPtr testObject;
3    //java.lang.String extends java.lang.Object
4    java::lang::SmtStringPtr testString (new java::lang::String("
         testing"));
5    //testObject gets a reference to the object that testString
         references
6    testObject = testString;
7    //now testString2 gets a reference to the object that
         testString references
8    java::lang::SmtStringPtr testString2 = testObject.Dynamic_cast
         ((java::lang::SmtStringPtr *) 0);
9    return 0;
10 }
```

4.10.2 instanceof

instanceof is a two-argument Java programming language keyword that tests whether the run-time type of its first argument is assignment compatible with its second argument[GJSB00]. In other words, **instanceof** is used to determine if one object belongs to the class hierarchy of a particular class. The C++ framework I built has a global **instanceof** function that works in a semantically identical manner to Java's **instanceof**.

4.11 Copy Construction and Reassignment

Because Java is a reference-based language any variable that is used is simply a reference to the underlying object. However, any time a Java reference is passed as an argument to a method it is a copy of the reference. Therefore, though Java is a reference based language, it is not a pass by reference language.

As an example of reference semantics, consider this program:

```
1  class Test{
2
3    public static void foo (StringBuffer stringBuffer_){
4      stringBuffer_.append("Tim");
5    }
6
7    public static void main (String [] _args){
8      StringBuffer stringBuffer = new StringBuffer ("I am ");
9
10     Test.foo (stringBuffer);
11     //at this point stringBuffer will equal "I am Tim"
12   }
13
14 }
```

Using the `SmartPtr` class that I provide the same semantics are preserved. For example, the same result occurs in the transformed C++ code below (reference counting

overhead code has been elided from the example below to make it more readable):

```
1  class Test;
2  typedef SmartPtr <Test> SmtTestPtr;
3  class Test : public Object {
4
5    public:
6      static void foo (SmtStringBufferPtr stringBuffer_) {
7          stringBuffer_->append("Tim");
8      }
9
10 };
11
12 int main () {
13     SmtStringBufferPtr stringBuffer (new StringBuffer ("I am "));
14     Test::foo(stringBuffer);
15     //at this point stringBuffer will equal "I am Tim"
16 }
```

4.11.1 Deep and Shallow Copies of Objects

In either C++ or Java some objects contain other objects and some of those contained objects contain objects themselves, etc. This capacity to contain an arbitrary depth of objects gives rise to terms like deep copies and shallow copies.

When an object (that contains a deep structure of other objects) is copied there is a choice of creating a deep or shallow copy of that object. A deep copy is a copy of an object that is an exact duplicate of the original such that any changes made to one of them will not result in changes to the other. At its most extreme, a shallow copy is an exact copy of the original object's pointers/references, such that the copy's contained pointers/references reference the same objects that the original object's refrence/pointers reference or point to. There are also hybrids of the above, that would also be referred to as a shallow copy. That is to say, if only some of the objects contained in a class were copied, while in other cases, just the pointers or references were copied, then it would be a shallow copy.

Copies of Objects in Java

In Java, when one object is assigned to another object, another reference to that object is created. That is to say, when one variable is assigned to another, a copy is not made. Copies of an existing object can only be made through the use of the keyword **new**. The use of this **new** keyword can be hidden by other methods (such as a clone method) or operators. For example, in the case of String objects, a new String object is constructed whenever the concatenation operator is used:

```
1 String string1 = new String("test");
2 String string2 = string1 + " complete";
3 //Now string1 contains the string "test" and string2 contains the
      string "test complete"
```

However, it is possible to create either a deep or shallow copy. For example, in the following a shallow copy is made:

```
1  class Test{
2
3    public static void main (String [] _args){
4      String string1 = new String ("I think, ");
5      String string2 = " therefore I am";
6
7      Vector vector1 = new Vector();
8      vector1.addElement(string1);
9      vector1.addElement(string2);
10     Vector vector2;
11     vector2 = (Vector) vector1.clone();
12     //vector2 and vector1 now reference the same string objects
13
14     String string1FromVector1 = vector1.get(0);
15     String string1FromVector2 = vector2.get(0);
16     //string1FromVector1 and string1FromVector2 reference the
           same object
17     if (string1FromVector1 != string1FromVector2){
18       //THE PROGRAM WILL NEVER REACH THIS BLOCK OF CODE
19     }
20   }
21
22 }
```

In the above example vector2 is a shallow copy of vector1. That is to say, the first element in vector1 actually references the same object that is referenced by the first element in vector2.

By making use of the smart pointer class, object assignment and copies can function in an identical manner to Java. The transform preserves the deep or shallow copy semantics.

4.12 Multiple Inheritance

Java does not support multiple inheritance. This forces programmers to create programs that lack some of the ambiguity that can be created in C++ programs[9]. In C++ both of the following two ambiguous architectures are possible:

- A class that extends multiple classes that each have a member function with identical function signatures/headers but with different implementations:

```
1  class B {
2    public:
3      void foo (){
4        //something
5      }
6  };
7
8  class C {
9    public:
10     void foo (){
11       //something else
12     }
13 };
14
15 class D : public B, public C {};
16
17 int main (){
18   D *pD = new D();
19
```

[9]Multiple inheritance does not always result in ambiguity - it some cases it enhances the architecture or structure of a program.

```
20    ((B*)pD)->foo();  // Use derivation through B
21    //or
22    ((C*)pD)->foo();  // Use derivation through C
23    //but
24    pD->foo();  // Invalid, doesn't compile - ambiguity
25    delete pD;
26    return 0;
27  }
```

- A class that extends multiple classes that all extend a common base class. This results in multiple instances of the same class being part of one object. Which therefore results in multiple "routes" to the base class if a cast is requested:

```
1   class A{};
2
3   class B : public A {};
4
5   class C : public A {};
6
7   class D : public B, public C {};
8
9   int main (){
10    D *pD;
11
12    (A*)(B*)pD;  // Use derivation through B
13    //or
14    (A*)(C*)pD;  // Use derivation through C
15    //but
16    (A*)pD;  // Invalid, doesn't compile
17  }
```

Instead Java allows the definition of either classes (abstract or not) or interfaces. So although Java does not allow a class to extend multiple classes, it can implement an arbitrary number of interfaces.

Java interfaces are used to define a collection of method definitions and constant values. It can later be implemented by classes that define this interface with the **implements** keyword[Sun02] or extended by other interfaces.

Since neither the generated code nor the base code to support the transformation make use of multiple inheritance the above ambiguity does not arise. However, in

creation of my C++ framework, I had to ensure that multiple inheritance was not used.

4.13 Difficulties/Limitations

There are many problems associated with doing a transformation from one language to another. Ideally in a transformation each statement can be transformed into a similar statement in the other language. However, there are issues that arise in a transformation from Java to C++ that are either non-trivial or simply not possible to transform. The only true limitations in the transform are imposed by the memory management strategy that was used in my implementation (smart pointer in conjunction with reference counting). Any Java code should be capable of being transformed into C++ code while maintaining the semantics of the original Java code (since both are Turing complete). However, since the garbage collection scheme being used in this C++ project is very rudimentary (reference counting used in conjuncture with smart pointers), problems with early and unexecuted deletions arise.

4.13.1 Access Level Difference Between Java And C++

Java supports the following four access levels on member variables within a class:

- **public** - Any class can access these members (variables, methods, classes).
- **protected** - Only the current class and classes that extend this class (inheritors) and/or any classes in this package can access these members.
- **private** - Only the current class can access these members.

- **default (nameless) package** - the default access specifier that is used if nothing is specified. Any classes that reside in the same packages as this class can access these members

One difference between the above and C++ is that protected access levels are not as "accessible" in C++ (i.e. in C++, classes in the same namespace can *not* access each others' protected members unless the **friend** keyword is used appropriately).

Another difference is that the **package** access level does not exist in C++. The **package** access level is specified by not prefixing the variable/class/method by any access level. For example, in the following on line 4 there is a member variable (`m_intVar3` that has the package access level):

```
1 class Foo {
2   public int m_intVar;
3   private int m_intVar2;
4   int m_intVar3;
5   protected int m_intVar4;
6 }
```

In C++, if no access specifier is given in a class, all members (variables, functions and inner classes) are private. However, there is another structure in C++ called a struct (derived from structure) and in this case all inheritance and members (variables, functions and inner classes) default to public. All Java classes and interfaces are transformed to C++ classes (not `struct`'s) by JCUV.

Two Possible Solutions to Resolving the Access Privilege Problem

1. Lose the finer grain access that Java provides and fall back to *public* - this would result in more classes having access to the members of a class than originally desired.

2. Use the **friend** qualifier between "accessor functions"[10] within classes thus allowing classes in the same namespace (analogous concept to Java's package) to access the member variables of a class through the accessor functions.

Therefore, since it will make cleaner and easier to understand C++ code, the former was chosen. That is to say, the C++ code will use *public* access in places where the Java code had *package* level access.

4.13.2 Unique Naming/Renaming

Java allows class members (methods, inner classes and variables) belonging to the same class to have the same name. This however is not possible in C++. Therefore, for a truly automated transformation from Java to C++ one would have to make use of a unique renaming of class members (functions, variables and inner classes) so that there are no naming conflicts between those members. This is not being done in this project so manual editing must be done to accommodate this.

4.13.3 String Usage

In Java it is possible to execute the following concatenations:

```
String tempString = "foo" + 42;
```

or

```
String tempString = 42 + "foo";
```

However, in C++, operator overrides can only be applied to user defined classes. So it is possible to allow the following types of concatenations in C++:

[10] get and set type functions in classes

```
String tempString1 = " foo";
String tempString2 = tempString1 + 42;
String tempString3 = tempString1 + " and another foo";
String tempString4 = 42 + tempString1 + " and one more";
```

However, the "+" operator cannot be overridden for two non user defined types (such as string and numerical literals). Therefore, string concatenation has to involve at least one user defined class object. To overcome this limitation, an additional transformation rule that searches for instances of concatenations of two primitive/integral data types could be added to JCUV. Once found these primitive/integral data types can be transformed into a String object or String objects. This is not currently being done.

4.13.4 Constructors

In Java, the programmer can invoke a different constructor of the same class or a base class on the first line of any of the other constructors. For example:

```
class Foo {
  private int m_id;
  public Foo (int _idValue){
    m_id = _idValue;
  }

  public Foo (){
    this (42);
  }

  public static void main (String [] args_){
    Foo fooObj = new Foo ();
  }

}
```

However, in C++, no constructor of a class can be invoked from any other constructor within the same class. However, providing a constructor has sufficient access

privileges, one class's constructor can invoke the constructor of any class it extends (directly or indirectly).

Therefore, the solution to this is to create a member function (in the generated C++ code) that contains the code that is being invoked by the other constructors so that the other constructors may call this member function. To do this in an automated manner would require the use of a unique naming scheme in order to ensure this new member function didn't conflict with the name of any existing member function in that class. However, this is currently not being done in an automated fashion.

4.13.5 Static Members of a Class

Java uses a period for access of either static or non-static members:

```
1 class Foo {
2   static int c_answer = 42;
3
4   public static void main (String [] _args){
5     Foo.c_answer = -42;
6   }
7 }
```

Whereas C++ uses two colons to access a static member in a class:

```
1 class Foo;
2 typedef SmartPtr <Foo> SmtFooPtr;
3 class Foo : public Object {
4
5   public:
6     static int c_answer;
7 };
8
9 int Foo::c_answer = 42;
10
11 int main (int argc, char * _args []) {
12     Foo::c_answer = -42;
13 }
```

Determination of static members across multiple Java classes is difficult because a single Java program can span an arbitrary number of files. Thus, each static member

variable encountered would have to be saved (to a database) between each execution of the TXL program on each file[11]. This transformation is currently not provided in an automated fashion. Therefore, manual editing is required everywhere a static access is made. However, currently a best effort approach is used that will change a period to a double colon if the member variable is static and contained within the current file.

4.13.6 Name Hiding and Scope

The C++ standard specifies that a member function which has been overloaded in a base class will not be visible. The only way to make it visible is to use the following syntax:

```
using class <base class>::<overloaded member function>;
```

For example:

```
1
2  class Object {
3    public:
4      virtual int answerToTheUltimateQuestion (){
5        return 42;
6      }
7  };
8
9  class DerivedClass : public Object{
10   public:
11     //the following line must be used to make the
           answerToTheUltimateQuestion function
12     //in Object visible to Derived class objects.
13     using Object::answerToTheUltimateQuestion;
14
15     virtual int answerToTheUltimateQuestion (int potentialAnswer)
16     {
17       if (potentialAnswer == 42){
18         return 1;
19       }
20       return 0;
```

[11] Another option would be to put all the Java files into one file and therefore a database would not have to be created.

```
21      }
22  };
23
24  int main (){
25
26    DerivedClass derivedClass;
27
28    int isSolutionCorrect = derivedClass.
          answerToTheUltimateQuestion(42);
29
30    int whatIsTheUltimateAnswer = derivedClass.
          answerToTheUltimateQuestion();
31    return 0;
32  }
```

In Java, this scope problem does not arise. Therefore, the `using` statement must be manually inserted into the generated C++ where applicable, so that member functions in any base class(es) are visible.

4.13.7 Class Definitions

In C++, sometimes a dependency will arise in the form of one class (A) using another class (B). But B has only had a forward definition and the class has not actually been defined yet. This is when member functions must be moved out of header files and into implementation files (i.e. make the member functions non-inline).

This change could be done in an automated fashion, but would require two runs of the TXL code over each Java file. One of the transforms would create the header file that contains the class definitions containing each of the function declarations, member variables and inner classes. Then the second transformation would create the implementation file that contains all of the function definitions.

Currently only the member functions that must be made non-inline are moved manually into an implementation file.

4.13.8 Nested/Inner Classes

Prior to the introduction of namespaces, nesting a class in C++ was an aid to name hiding and code organization. In addition, Java packaging provides the equivalent of namespaces, through the use of packages.

A nested class in Java simply means that a class that is defined within another class. An inner class is a specific type of a nested class. An inner class is a non-static nested class. In Java, an inner class object keeps a handle to its outer class object. The inner class object may access members of the outer class object without qualification, as if those members belonged directly to the inner class object. In addition, the inner class has access to any members (private or otherwise) of the outer class and vice versa. This provides a much more elegant solution to the problem of callbacks, solved with pointers (or references) to members in C++.

In order to provide this functionality, any inner classes in the generated C++ code must be given a reference to their outer class in any of their constructors and they must be made a friend of the outer class. Then any unqualified calls to member functions in their containing class must be prefixed by their reference to the containing class. For example the following Java code:

```
1  class Foo {
2    private int m_intVar;
3    class InnerFoo {
4      public InnerFoo (){
5      }
6
7      public void Test (){
8        m_intVar = 42;
9      }
10   }
11 }
```

would be transformed to the following C++:

```
1  class Foo {
```

```
2   private:
3     int m_intVar;
4
5   public:
6     class InnerFoo {
7       Foo & m_outerClassRef;
8
9       public:
10        InnerFoo (Foo & foo_):m_outerClassRef(foo_){
11        }
12
13        void Test (){
14          m_outerClassRef.m_intVar = 42;
15        }
16    };
17
18  public:
19    friend class InnerFoo;
20
21 };
```

4.13.9 Static Virtual Members of Classes

Java supports static virtual members (every member of a class in Java is considered virtual) of classes, but C++ does not. The only time this is relevant is if a static member is invoked/accessed through an object instead of a class. For example in Java:

```
1  class Foo {
2    public static int c_test = 42;
3  }
4
5  class DerivedFoo extends Foo {
6    public static int c_test= 9283;
7
8    public static void main (String [] args_){
9      DerivedFoo derivedFoo = new DerivedFoo();
10     //the following line will print 9283
11     System.out.println (derivedFoo.c_test);
12     Foo foo = new Foo();
13     //the following line will print 42
14     System.out.println (foo.c_test);
15   }
16 }
```

4.13.10 Inherent Weakness of Reference Counting Memory Management Strategy

Early Deletions

If the **this** keyword is used in the constructor and bound to a smart pointer the object will end up deleting itself before the constructor completes. The reason this occurs is because in the constructor nothing has a reference to the object represented by **this** yet. Therefore, when the object that is bound to **this** goes out of scope (i.e., before the constructor finishes) the reference count will hit zero and the object will delete itself before it returns from its constructor.

Unexecuted Deletions

Sun's Java Virtual Machine is able to handle not only self references but convoluted circular dependencies through the use of Sun's advanced Garbage Collection mechanism[Ven03]. However, since the smart pointer class being used in the C++ code relies on use of a very basic mechanism (namely reference counting) it is not possible to provide full support for circular dependencies.

There are ways in which circular references could be supported, but it would involve manual customization of the class in question. For instance, the following class has a circular dependency on itself that will result in memory never being reclaimed (should this class ever be instantiated) if the member variable `testPtr` is never reassigned to something other than **this**:

```
1 class Test;
2 typedef SmartPtr<Test> DynTestPtr;
3 class Test {
4   DynTestPtr m_testPtr;
5
```

```
6    Test () :m_testPtr(this){}
7
8  };
```

However, the following code would result in memory being reclaimed, but forces the user to pick and choose when smart pointers are applicable instead of using them in all cases:

```
1  class Test;
2  typedef SmartPtr<Test> DynTestPtr;
3  class Test {
4    Test * m_testPtr;
5
6    Test () :m_testPtr(this){}
7
8    ~Test(){
9      delete m_testPtr;
10   }
11 };
```

4.13.11 Import Statements of Entire Packages

In Java it is possible to import an entire package of classes. However, in C++ only a single class can be included per include statement. Therefore, the following statement in Java:

```
import java.util.*;
```

can not be transformed into an equivalent C++ **include** statement. So manual editing is performed to determine which classes in a particular package are actually be imported and then a single include statement must be used for each individual file to be included. However, an equivalent namespace statement can be used as shown in Section 4.4.

4.14 Summary

This chapter illustrated the differences between Java and C++ and how I performed the transformation from Java to C++. Of most importance is the memory management differences between Java and C++. Another important issue is that there are Java keywords that don't exist in C++. In most cases, a coding mechanism can be used to emulate the functionality of these keywords. One of the interesting concepts that required transform from Java to C++ was emulating the functionality of Java arrays in C++. Lastly, there were a number of limitations that arose that required non-trivial solutions (for instance a unique naming/renaming scheme).

The next chapter will explain the details of the second step of the implementation illustrated in Figure 3.1

Chapter 5

Step Two: C++ Using RMI to C++ Using Verisoft

The second step of my solution required generation of the Naming, UnicastRemoteObject and the remote stub for the user defined remote class. In addition, currently the main entry method for the program to be analyzed by Verisoft must be manually created. Lastly, the limitations of this step are discussed.

5.1 Generation of Naming

As illustrated in Figure 2.1, in Java, the Naming class acts as a local interface for the potentially remote RMIRegistry. Naming can be used initially to add a remote object to the RMIRegistry. After a remote object has been added to the RMIRegistry, a local object can get a reference to the proxy for the remote object.

In my implementation, the RMIRegistry is a hashtable object in shared memory. After the designated remote object is added to the shared memory RMIRegistry, the local object can then retrieve a proxy for that remote object. Then invocations on the proxy object will work in a similar manner to the Java RMI framework.

One part of the second step involved the generation of the Naming class (which also contains a modelled version of the RMI registry). This was required as the C++ code that is being used does not support functionality similar to Java's reflection library. Java's reflection library allows a developer to obtain reflective information about classes. As it applies in this situation, it is the ability to instantiate a class whose name is not known until run-time. That is to say, to instantiate an object based on a string containing its class name.

Therefore, to model this behaviour in C++ the lookup method is generated, so that the remote class's stub is returned based on the string that is passed into the lookup method of Naming.

Here is an example of a generated lookup method in Naming (using the deadlock code originally from Figure 1.1 and then transformed into Figure 4.2):

```
1  static SmtRemotePtr lookup (SmtStringPtr name_)
2  {
3    SmtObjectPtr object = c_hashtable->get (name_);
4    if (object != '\0') {
5      if (instanceOf (object, PeerBInterface)) {
6        return SmtPeerB_StubPtr (new PeerB_Stub);
7      }
8    }
9    return '\0';
10 }
```

The above code attempts to find an object that is associated with the `name_` parameter. If it manages to find an object that is associated with the `name_` parameter it returns an instance of an object of that class with the same name plus

a "_Stub" appended to it. If no such object exists in the hashtable, then null is returned.

5.2 Generation of Remote Object Stub (Proxy For Remote Object)

The stub class (proxy for the remote object) is generated such that it contains methods with the same signature (return type, name and parameters) but its contents actually send messages to the true remote object and waits for an acknowledgement (i.e. the methods are blocking). This requires replacing all internet communication in RMI by Verisoft methods that make use of inter-process communication.

The following code was also generated using the deadlock example originally from Figure 1.1:

```
1  virtual void executeTask ()
2  {
3    char * message = new char [100];
4    sprintf (message, GLOBAL_executeTask_VAR);
5    send_to_queue (m_remoteObjectMsgQueueID, QSZ, message);
6    delete [] message;
7    message = (char *) rcv_from_queue (m_msgQueueID, QSZ);
8    if (strcmp (message, GLOBAL_executeTask_VAR) == 0) {
9      //I have received the ack I was waiting for - the method I
           called completed
10     return;
11   }
12   throw SmtRemoteExceptionPtr ("problem in transmission of
       message");
13 }
```

The above code snippet puts the string of characters that are represented in the macro `GLOBAL_executeTask_VAR` into the message character array. Then it sends those characters to the messaging queue variable named `m_remoteObjectMsgQueueID`. The messaging queue object is a Verisoft specific class that is used to send messages

between processes. Once those characters have been sent to the queue, the space for the characters is deleted locally. Then the code blocks until it receives a message from the queue. If the message is what it expects (i.e. the same message it sent to the queue) it will return without exception. Otherwise the smart-pointer version of a `RemoteException` object will be thrown.

5.3 Generation of UnicastRemoteObject

All remote objects in the Java RMI framework must extend UnicastRemoteObject. The generation of the UnicastRemoteObject is necessary to accept the incoming messages from the stub class, invoke the appropriate method, and then send a message back to the stub class indicating the method has completed. In some Java RMI applications, parameters are sent to the remote object and an object is returned from the remote object's method. In order for this to occur, objects must be marshalled and unmarshalled. Marshalling an object is the process of creating a byte array that represents that object; then unmarshalling is using that byte array to reconstruct the object. However, JCUV only supports the marshalling and unmarshalling of integer data type objects as outlined in Section 5.5.2.

Using the same deadlock example, here is part of the run method for the `UnicastRemoteObject`:

```
1  void run ()
2  {
3    char * message;
4    while (1) {
5      message = (char *) rcv_from_queue (m_msgQueueID, QSZ);
6      if (strcmp (message, GLOBAL_executeTask_VAR) == 0) {
7        this->executeTask ();
8        send_to_queue (m_remoteObjectMsgQueueID, QSZ, message);
9      }
10   }
```

```
11 }
```

The above code snippet enters into an infinite loop (a simplification of what is done in Java) and waits for messages to be sent to it. If the message is the string represented by the `GLOBAL_executeTask_VAR` macro, then it will invoke the current object's executeTask method. Then the code will block until the executeTask method finishes, then it will send a message back to the message queue indicating the method has completed.

5.4 Manual Creation of Main Entry Function

Since the original Java RMI program had 2 programs (with two separate main entry functions) the new C++ program will also have two main entry functions. This requires the two main functions to be renamed such that their signature is not either:

```
1 int main() { /* ... */ }
```

or

```
1 int main(int argc, char* argv[]) { /* ... */ }
```

In addition, a new main function must be created that creates each of the processes that will be run. This requires use of the Unix/Linux `fork` command. Then once both of the processes have been `fork`'ed, the respective transformed main methods from the original Java program can be executed.

5.5 Difficulties/Limitations

5.5.1 RMI Server Class Constraints

As previously mentioned (in Section 2.7.2), there are two primary ways of creating a Java RMI server object. The current implementation of this transformation only supports a Java RMI server class that both implements an interface that extends the java.rmi.Remote interface and directly extends java.rmi.server.UnicastRemoteObject.

5.5.2 Marshalling/Unmarshalling of Objects

In some Java RMI applications, parameters are sent to the remote object and an object is returned from the remote object's method. In order for this to occur, objects must be marshalled and unmarshalled. Marshalling an object is the process of creating a byte array that represents that object, then unmarshalling is using that byte array to reconstruct the object. Currently the only parameter or returned object that can be marshalled or unmarshalled is the integer data type. This subsequently reduces the set of programs that can be transformed.

5.5.3 Lack of Java Reflection Functionality

Ideally, the C++ framework will support a reflection concept in an identical manner to the Java reflection framework. This would allow objects to be instantiated at run-time based upon a string that represents their class name. Currently this is overcome by generating code in a class to substitute for the functionality provided in the Java reflection framework.

5.6 Summary

This chapter has given an overview of the second step of my implementation. The generation of the UnicastRemoteObject and the Naming class was necessary only because currently my implementation does not support the functionality provided in Java's reflection framework. In addition, just like Java, the remote stub for the user defined remote object must be generated prior to run-time. Lastly, the limitations of this approach were discussed.

The next chapter will address the third and last step of my implementation. This involves compiling and then linking the C++ code, configuring Verisoft for the analysis of interest. Lastly the limitations of that step are discussed.

Chapter 6

Step Three:

Analysis Using Verisoft

Currently, the main method for the entire program must be manually created. This allows the user to specify the number of client/server or peer objects that should be created in the program.

6.1 Compiling/Linking

The last step simply involves compiling and linking the relevant C++ files and then executing the resultant executable using Verisoft.

6.2 Pre-Run-Time Details

Before executing the resultant executable the system_file.VS file of Verisoft should be configured appropriately. Factors such as the number of processes that will execute,

the analysis depth (i.e. how deep in the state space Verisoft will traverse before it stops executing), whether to ignore deadlocks, and more must all be specified in the system_file.VS file prior to execution of Verisoft. Lastly, the system_file.VS must be in the same directory as the a.out executable (generated from compiling and linking the C++ source code).

Verisoft can be used in one of three modes (manual, guided, or automatic simulation mode) to analyze the resulting model. Manual mode is where the user will manually step through the execution of the code. Guided mode is used if the user wants to specify when a particular process will execute its next visible operation or which number is chosen at a toss point (a point in the code where a range of numbers can be selected to be returned from a function). Automatic mode allows Verisoft to run automatically and return at what point (if any) in the state space of the program execution that it found a deadlock, divergence or livelock.

6.3 Limitations

6.3.1 Specific to Verisoft

Currently, Verisoft requires that the number of processes be specified in the system_file.VS file before Verisoft is executed. Thus, dynamic process creation at runtime is not supported.

In addition, the executable (generated from compiling and linking the C++ source code) has to be renamed a.out to allow Verisoft to analyze the application. If the application file (executable) is not renamed to a.out the user will receive the error message “Error 2 in ftok (getkey): No such file or directory”.

6.3.2 Specific to JCUV

The largest limitation specific to JCUV is the requirement that the user understands how to create C++ makefiles. Thus even after the appropriate C++ files have been transformed and/or generated, the user must still create the compile and link those C++ files.

Chapter 7

Results

7.1 Experiments/Results

The complexity of any of the transformations (in either step 1 or 2) is O(n).

java.util.Hashtable and its dependent classes (over 14,000 lines of code) were transformed and tested to ensure the behaviour matched that of the original Java java.util.Hashtable. One hundred different C++ tests were written and executed successfully.

So far only two small RMI examples have successfully been transformed. Each program makes use of the transformed java.util.Hashtable and its dependent classes. As a consequence each of the RMI programs has a total source code of greater than 14,000 lines of code. The generated C++ source code has a similar number of lines of code (though the number of lines differs slightly due to whitespace) as the original Java code.

In the example provided in Figure 1.1 the transformation for the non-implicit classes (i.e. PeerA and PeerB classes) was completed within 5 seconds. The analysis

```
[cassidy@txl simpleRMI]$ verisoft main.c
g++ -I/home/cassidy/verisoft/bin /home/cassidy/verisoft/bin/verisoft_Linux_2.x.o -DVERIFY main.c

VeriSoft Version 2.0.6:
      search for deadlocks, livelocks, divergences, and assertion violations

state space saved in file sss.VS

intermediate report: depth is set from 5 to 10, current breadth is 1

deadlock detected at depth 7 !!!
error trace saved in error1.path

Number of states: 7
Number of transitions: 7

VeriSoft Version 2.0.6
0.02user 0.08system 0:00.11elapsed 86%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (424major+4792minor)pagefaults 0swaps
```

Figure 7.1: Output from trivial deadlock example

by Verisoft (in which deadlock was found) also completed within 5 seconds.

Figure 7.1 shows the output from Verisoft being run in automatic mode on the trivial deadlock example originally derived from the code in Figure 1.1.

If used in guided or manual simulation mode, the Verisoft tool will produce a graphical representation of the state space of a program in the form of a tree. In the trivial deadlock example the tree has no branches. However, in more complex programs (programs with a greater number of states) the tree branches off at an exponential rate.

In another example, the program was a very simplified version of a financial transaction system. The clients all shared the same account and would deposit money into the shared account and get their balance. This example made use of 10 clients making remote method invocations on a remote object. In one execution a divergence was successfully found at a depth of eight visible operations. The transformation in

this case also took under five seconds, however, the actual analysis by Verisoft took approximately 10 seconds.

Chapter 8

Conclusion

8.1 Future Work

Currently, there are a number of a limitations which it will be feasible to address in the near future while others are much more complex. The easiest limitation to address would be Class Definitions (from Section 4.13.7 in step one) while the most difficult would be creating a better garbage collection algorithm (as mentioned in Section 4.13.10).

The essence of JCUV is concurrency analysis of Java code. The eventual goal would be to analyze any form of concurrency implemented in Java. This includes, but is not limited to, more generalized use of the Java RMI framework (i.e., with support for marshalling and unmarshalling of more than just the integer data type), internet communication without the use of RMI and even simple thread communication.

So far, the approach has only been applied to relatively small Java RMI applications. Future work will also attempt to use JCUV on large and more realistic pieces of Java RMI code.

Though not currently being done, a transform back to the original Java code should be possible by reversing the application of the rules generated in the TXL program. However, this becomes more complicated if any manual editing has been done after the automated transformation has completed. That is to say, the transformation back to the originating Java code should not be difficult so long as the generated C++ code is not modified manually. This is useful as it allows the analysis output like error traces or counterexamples to be mapped back to the source code automatically.

Another possibility for future work would be to use the transformational software and the C++ framework I have written to perform a fully automated semantics preserving transformation from Java to C++.

8.2 Conclusions

Concurrency in any program can be a source of intermittent problems which thus makes these problems very difficult to debug. Tools like Bandera[HD01] and Java PathFinder[Hav99] are useful in transforming source code into a modelling language. The subsequent model can then be analyzed by the modelling tool.

However, it is difficult and sometimes simply intractable to attempt a transformation from a programming language to a modelling language. This problem is better known as the Model Construction problem, which Verisoft addresses extremely well since Verisoft is able to analyze C or C++ code directly.

JCUV provides the user with an approach to expand the number of programming languages that Verisoft is capable of supporting. JCUV is a three step transformation from Java code that makes use of RMI to C++ code that uses Verisoft.

The first step is a semantics preserving transformation from Java to C++. To support this transformation a C++ framework had to be built to support concepts such as automated memory management and arrays that functioned like Java arrays. In addition, a number of mechanisms in Java had to be emulated in C++, such as the java keywords **instanceof** and **synchronized**. In this process, numerous limitations were encountered. However, in the opinion of the author, all of these limitations can be addressed and surmounted in an automated fashion.

The second step involves the generation of C++ classes that make use of Verisoft instead of Java RMI classes. Essentially the interface of all of the RMI classes remain, but the implementation is changed to make use of Verisoft functionality. The greatest limitation at this stage was the lack of support for marshalling/unmarshalling of code.

The third step involves compiling and linking the resultant C++ code, then configuring and running Verisoft. This typically requires the creation of a C++ makefile which, depending on the size of the code can be quite large. After which, Verisoft can be run in either manual, automated or guided simulation.

Despite a number of limitations, I consider the work presented in this dissertation a very promising first step towards leveraging the benefits of Verisoft and TXL for concurrency analysis.

Bibliography

[BCM+90] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, 1990.

[BD00] Bernd Bruegge and Allen H. Dutoit. *Object-Oriented Software Engineering.* Prentice Hall, New Jersey, 2000.

[BDHS00] Dragan Bosnacki, Dennis Dams, Leszek Holenderski, and Natalia Sidorova. Model checking SDL with Spin. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 363–377, 2000.

[Bru03] Bruce Eckel. Comparing C++ and Java. Web, 2003. `http://www.javacoffeebreak.com/articles/thinkinginjava/comparingc++andj%ava.html`.

[CDH+00] James Corbett, Matthew Dwyer, John Hatcliff, Corina Pasareanu, Robby Shawn Laubach, and Hongjun Zheng. Bandera: Extracting finite-state models from java source code. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 439 – 448, Limerick, Ireland, June 2000. ACM Press.

[CDH01] James Corbett, Matthew Dwyer, and John Hatcliff. Expressing Checkable Properties of Dynamic Systems: The Bandera Specification Language. *Robby KSU CIS Technical Report 2001-04*, June 2001.

[CDMS01] James R. Cordy, Thomas R. Dean, Andrew J. Malton, and Kevin A. Schneider. Software Engineering by Source Transformation - Experience with TXL. *IEEE 1st International Workshop on Source Code Analysis and Manipulation*, pages 168–178, November 2001.

[CDMS02] James R. Cordy, Thomas R. Dean, Andrew J. Malton, and Kevin A. Schneider. Source Transformation in Software Engineering Using the TXL Transformation System. *Journal of Information and Software Technology*, 44(13):827–837, October 2002.

[CGP02] S. Chandra, P. Godefroid, and C. Palm. Software Model Checking in Practice: an Industrial Case Study. In *Proceedings of International Conference on Software Engineering*, Orlando, May 2002.

[Cor02] Rational Software Corporation. *Rational Quality Architect Realtime Edition Users Guide*, May 2002. VERSION: 2002.05.20.

[GHJ98] P. Godefroid, R. Hanmer, and L. Jagadeesan. Systematic Software Testing using VeriSoft: An Analysis of the 4ESS Heart-Beat Monitor. *Bell Labs Technical Journal*, 3(2), April 1998.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, Boston, 1995.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Sun Microsystems, 2000.

[GK00] Orna Grumberg and Shmuel Katz. Faithful Translations Among Models and Specifications in VeriTech. *Israel Institute of Technology*, May 2000. `http://www.cs.technion.ac.il/Labs/ssdl/research/veritech/index.html`.

[God96] Patrice Godefroid. On the Costs and Benefits of using Partial-Order Methods for the Verification of Concurrent Systems. *Proceedings of DIMACS Workshop on Partial-Order Methods in Verification*, July 1996.

[God01] Patrice Godefroid. *Verisoft Reference Manual*. Bell Laboratories, Lucent Technologies, February 2001.

[Hat01] John Hatcliff. Specification and verification of reactive systems. Class Lecture/Slides, 2001. `http://www.cis.ksu.edu/~hatcliff/842/Slides/SPIN-Temporal-Logic.pdf`.

[Hav99] K. Havelund. Java PathFinder: A Translator from Java to Promela. *Theoretical and Practical Aspects of SPIN Model Checking – 5th and 6th International SPIN Workshops*, 1680:152, 1999.

[HD01] John Hatcliff and Matthew Dwyer. Using the Bandera Tool Set to Model-check Properties of Concurrent Java Software. *Proceedings of CONCUR 2001*, pages 1–10, June 2001.

[Hol97a] Gerard J. Holzmann. *Basic Spin Manual*, August 1997. `http://spinroot.com/spin/Man/Manual.html`.

[Hol97b] Gerard J. Holzmann. The Model Checker Spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, 1997.

[HP00] Klaus Havelund and Tom Pressburger. Model Checking Java Programs Using Java PathFinder. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4), April 2000.

[HS02] Gerard Holzmann and Margaret Smith. An Automated Verification Method for Distributed Systems Software Based on Model Extraction. *IEEE Trans. on Software Engineering*, 28(4):364–377, April 2002. `http://spinroot.com/spin/Doc/ieee97.pdf`.

[HT01] John Hatcliff and Oksana Tkachuk. *The Bandera Tools for Model-checking Java Source Code: A User's Manual*. Bell Laboratories, Lucent Technologies, March 2001. `http://www.cis.ksu.edu/~santos/bandera/tut/doc/tutorial.html`.

[IDH03] Radu Iosif, Matthew B. Dwyer, and John Hatcliff. Translating Java for Multiple Model Checkers: the Bandera Back-End (extended version). *SAnToS technical report SAnToS-TR2003-4*, September 2003. `http://www.cis.ksu.edu/~hatcliff/Temp/FMSD-iosif-dwyer-hatcliff.ps.2.ps%`.

[Inf99] InfoStreet, Inc. Instantweb: Online computing dictionary. Web, 1999.

[Int98] International Standards Organization, International Electrotechnical Commission, American National Standards Institute, Information Technology Industry Council. *Programming languages - C++*. American National Standards Institute, first edition, September 1998.

[Jon86] Capers Jones. *Programming Productivity.* McGraw-Hill Companies, Inc., New York, 1986.

[Kal99] Danny Kalev. *ANSI/ISO C++ Professional Programmer's Handbook.* Macmillan Computer Publishing, 1999.

[LAH+99] Jesse Liberty, Vishwajit Aklecha, Steve Haines, Steven Mitchell, Alexander Nickolov, Charles Pace, Meghraj Thakkar, Michael J. Tobler, Donald Xie, and Steve Zagieboylo. *C++ Unleashed.* Sams, Indianapolis, Indiana, 1999.

[McC96] Steve McConnell. *McConnell (RD) on QA.* Microsoft Press, 1996.

[McM92] K. L. McMillan. Symbolic Model Checking - An Approach to the State Explosion Problem. *PhD thesis: Carnegie Mellon University*, June 1992.

[McM93] K.L. McMillan. Symbolic Model Checking: An Approach to the State Explosion Problem. *Kluwer Academic Publishers*, 1993.

[McM00] K.L. McMillan. *The SMV System.* Model Checking Project, Carnegie Mellon, November 2000. `http://www-2.cs.cmu.edu/~modelcheck/smv/smvmanual.ps`.

[MJ97] J.M.R. Martin and S.A. Jassim. A Tool for Proving Deadlock Freedom. In Proceedings of WoTUG 20 (World occam and Transporter User Group), editor, *Parallel Programming and Java*, volume volume 50 of Concurrent Systems Engineering, pages 1–16, University of Twente, Netherlands, April 1997. IOS Press.

[Pre97] Roger S. Pressman. *Software Engineering: A Practitioner's Approach.* McGraw-Hill Companies, Inc., New York, 1997.

[Sch96] Klaus Schneider. CTL and Equivalent Sublanguages of CTL. *Chapman and Hall*, 1996. `http://goethe.ira.uka.de/~schneider/my_papers/Schn97a.ps.gz`.

[Sel03] Bran Selic. Bubbles of Steel: A Preview of UML 2.0 and MDA. Seminar, 2003.

[SM00] P.H. Shiu and V. J. Mooney. The Principle of Parallel Deadlock Detection. , December 2000. `http://citeseer.nj.nec.com/542576.html`.

[Sto03] Scott Stoller. Stony Brook University. Personal Communication, 2003.

[Str97] Bjarne Stroustrup. *The C++ Programming Language, Third Edition*. Addison-Wesley, New Jersey, 1997.

[Str03] Bjarne Stroustrup. Texas A & M University . Personal Communication, 2003.

[Sun01] Sun Microsystems, Inc. Java Remote Method Invocation. Web, 2001. `http://java.sun.com/j2se/1.4.1/docs/guide/rmi/spec/rmiTOC.html`.

[Sun02] Sun Microsystems, Inc. JavaTM 2 Platform, Standard Edition, v 1.4.0 API Specification. Web, 2002. `http://java.sun.com/j2se/1.4/docs/api/index.html`.

[Sun03a] Sun . Implementing nested classes. Web, 2003. `http://java.sun.com/docs/books/tutorial/java/javaOO/nested.html`.

[Sun03b] Sun Microsystems, Inc. Lesson: All About Sockets . Web, 2003. `http://java.sun.com/docs/books/tutorial/networking/sockets/index.html`.

[Sun03c] Sun Microsystems, Inc. Threads: Doing Two or More Tasks At Once. The Java Tutorial: A Practical Guide For Programmers. Web, 2003. `http://java.sun.com/docs/books/tutorial/essential/threads/`.

[TXL02] TXL Software Research Inc. About TXL. Web, 2002. `http://www.txl.ca/nabouttxl.html`.

[Ven03] Bill Venners. *Inside the Java Virtual Machine*. Artima Software, Inc., 2003.

[Vis01] Eelco Visser. A Survey of Strategies in Program Transformation Systems. *Electronic Notes in Theoretical Computer Science*, 57:363–377, 2001.

[Wal03] Todd Wallentine. Kansas State University, Kansas. Personal Communication, 2003.

[Wha01] Whatis.com. Port: [definitions]. Web, July 2001. `http://searchnetworking.techtarget.com/sDefinition/0,,sid7_gci212807,00%.html`.

[Wha02] Whatis.com. Connectionless and Connection-Oriented: [Definitions]. Web, Oct 2002. `http://searchnetworking.techtarget.com/sDefinition/0,,sid7_gci856314,00%.html`.

Appendix A

Code Examples

A.1 Java RMI Exception To java.rmi.Remote Extension

There is an exception to the above rule that does not require that the class or interface that extends the java.rmi.Remote interface to contain all of the methods that can be called remotely:

The following shows a valid remote interface Beta that extends a non-remote interface Alpha, which has remote methods, and the interface java.rmi.Remote:

```
1  public interface Alpha {
2      public final String okay = "constants are okay too";
3      public Object foo(Object obj)
4          throws java.rmi.RemoteException;
5      public void bar() throws java.io.IOException;
6      public int baz() throws java.lang.Exception;
7  }
8
9
10 public interface Beta extends Alpha, java.rmi.Remote {
11     public void ping() throws java.rmi.RemoteException;
12 }
```

[Sun01]

A.2 C++ Going Out of Scope Example

Going out of scope simply means that the flow of control in a program has left a block of code in which an object was created and thus no longer is accessible. For example:

```
1  int main (){
2    if (true){
3      char tempChar = 'a';
4    }
5    //tempChar has gone out of scope
6    {
7      int tempInt = 3;
8    }
9    //tempInt has gone out of scope
10   return 0;
11 }
```

A.3 C++ Smart Pointer Class

```
1  template<class DynObject>
2  class SmartPtr
3  {
4    public:
5      DynObject *m_dynObjPtr;
6
7    public:
8      SmartPtr();
9
10     SmartPtr( DynObject *_dynObjPtr );
11
12     //<necessary for polymorphic assignment>
13     template<class O>
14     SmartPtr(SmartPtr<O> const & other);
15     //</necessary for polymorphic assignment>
16
17     //need the following copy constructor so that VC++ doesn't
           complain
18     SmartPtr (const SmartPtr & _smartPtr);
19
20     SmartPtr (const char * charPtr_);
```

```
21
22      SmartPtr (const int * intVar_);
23
24      virtual ~SmartPtr();
25
26    // Extractors
27    public:
28      // Get underlying pointer for method calls
29      DynObject* operator->();
30
31    // Assignment
32    public:
33      SmartPtr & operator=( const SmartPtr & _dynObject);
34
35      //<necessary for polymorphic assignment>
36      template<class O>
37        const SmartPtr<DynObject>& operator=(SmartPtr<O> const &
              other);
38      //</necessary for polymorphic assignment>
39
40      //added explicitly for Vector usage
41      int operator<(const SmartPtr & _dynObject);
42
43      //added explicitly for Vector usage
44      int operator== (const SmartPtr & _dynObject);
45
46      //Recall that the parameter being passed into this function
           will NOT be used
47      //The only reason it is necessary to include it at all is so
           VC++ will support this
48      //use of templates
49      template <class U>
50        SmartPtr<U> Dynamic_cast(U * _smartPtr);
51
52      //<needed for String operations>
53      operator char*() const;
54
55      DynObject * operator+=(const char * _char);
56
57      SmartPtr<DynObject> operator+(const char * _char);
58
59      SmartPtr<DynObject> operator+(const SmartPtr & _dynObject);
60
61      SmartPtr<DynObject> operator+(const int intVar_);
62      //</needed for String operations>
63
64    // Content testing
65    public:
66      bool operator!();
67
68      operator bool();
```

```
69
70    // Implementation
71    protected:
72      void Release();
73
74      void Store( DynObject *_dynObjPtr );
75  };
```

A.4 Mundane C++ Static Array Class

As is evident from the example below, the declaration of the array and the definition of its size must occur within the same expression. Unlike in Java, the definition of the size of the array can be defined in a completely different expression as the one that declares that array.

```
1
2   int main (){
3     //At the point of the array declaration - the size of the array
          must be defined
4     //In this case, the array is decided to range from a starting
          value of 0 to a value of 9
5     CStaticArray <int, 0, 9> intArray;
6     //Assignment to an element in the array works as would be
          expected of an array
7     DynTestPtrArray [0] = 3;
8
9     return 0;
10  }
```

A.5 std::vector Wrapper class

```
1   template <class VectorElement>
2   class StdVector : public Object
3   {
4     private:
5       std::vector<VectorElement> m_stdVector;
6
7     public:
8       StdVector () :length (m_stdVector);
```

```
9
10      explicit StdVector(std::vector<VectorElement>::size_type size
            )
11      :m_stdVector(size), length (m_stdVector);
12
13      StdVector(const StdVector& rhs)
14      :m_stdVector(rhs.m_stdVector), length(m_stdVector);
15
16      StdVector& operator=(const StdVector& rhs);
17
18      class Length {
19          std::vector<VectorElement> & m_vector;
20      public:
21          Length(std::vector<VectorElement>& vec) : m_vector(vec);
22
23          operator int();
24
25      } length;
26
27      StdVector* operator->();
28
29      VectorElement& operator[](std::vector<VectorElement>::
            size_type index_);
30
31      int operator== (const StdVector & stdVector_);
32
33      int operator== (const int intVar_);
34
35  };
```

Appendix B

Rules for Volatile Variables

If a variable is declared volatile, then additional constraints apply to the actions of each thread. Let T be a thread and let V and W be volatile variables.

A use action by T on V is permitted only if the previous action by T on V was load, and a load action by T on V is permitted only if the next action by T on V is use. The use action is said to be "associated" with the read action that corresponds to the load. A store action by T on V is permitted only if the previous action by T on V was assign, and an assign action by T on V is permitted only if the next action by T on V is store. The assign action is said to be "associated" with the write action that corresponds to the store. Let action A be a use or assign by thread T on variable V, let action F be the load or store associated with A, and let action P be the read or write of V that corresponds to F. Similarly, let action B be a use or assign by thread T on variable W, let action G be the load or store associated with B, and let action Q be the read or write of

W that corresponds to G. If A precedes B, then P must precede Q. (Less formally: actions on the master copies of volatile variables on behalf of a thread are performed by the main memory in exactly the order that the thread requested.) The load, store, read, and write actions on volatile variables are atomic, even if the type of the variable is double or long [GJSB00].

Vita

Tim Cassidy

1643 Sunnyside Road
Kingston, Ontario
K7L 4V4

cassidy@cs.queensu.ca

http://www.cs.queensu.ca/~cassidy

Summary of Skills

- Operating Systems: AIX, HP, and Solaris versions of Unix, Windows 3.x/95/98/NT/2000, DOS and Mac OS 8.1
- Programming Languages: C, C++, Java using JDK 1.1.7, 1.1.8, and 2 (version 1.2 and 1.3), Javascript, Perl version 5.003, Korn Shell scripting, Pascal, Object Oriented Turing, Machine Code, Assembly Language, Miranda, Prolog, and LISP
- Databases: Oracle 8.0.4 using SQL 8.0, Microsoft SQL Server 7.0 and DB2
- Programming Applications: ClearCase Revision Control System, Microsoft Visual C++ 6.0, Orbix 2.3c02, OrbixWeb versions 2.0.1 to 3.1, Live Software's JRunPro 2.3.1 Servlet Runner, Unify eWave

ServletExec 3.0, Microsoft Visual J++ 1.1 and 6.0, Rational Rose 2000 Professional J Edition, Rational Rose Enterprise Edition, Rational Rose RealTime, INTERSOLV PVCS Version Manager, and Microsoft Visual SourceSafe

- Servers: Java Web Server 1.1.3, Netscape's Enterprise Server, Netscape's Directory Server and Internet Information Server
- Administrative: Technical documentation (UML, Functional Specifications) and non technical documentation (users' manual and help pages), collaborative data collection, ability to develop new programs (three industry based and two personal)

Experience

- **Software Designer**
  June 2001 to May 2002
  Entrust, Inc.
  Ottawa, Ontario
  - Use of ClearCase Revision Control System as a source control (configuration management) tool.
  - Wrote functional specification and used UML to model software to be designed.
  - Low level C and object oriented C++ cryptography toolkit design.
  - Understanding of compatibility issues using C++ with Windows NT/2K, AIX, HP-UX, and Solaris.
  - Implemented OCSP (RFC 2560) component.
  - Gained extensive knowledge of PKI.
  - Took Entrust's "Entrust Authority Administrator Training" course.
- **Programmer**
  September 2000 to May 2001
  MediaShell
  Kingston, Ontario
  - Programming in Java (specifically Applets and stand-alone applications using Swing).
  - Creating signed applets specific to Netscape, IE and to the Java plug-in using a test certificate as well as a certificate from a Certificate Authority.

- Working in Perl, specifically as it applies to the e-commerce application Interchange (created by an organization called Akopia).

- **Software Developer**
  May 1999 to August 2000
  The Bulldog Group
  Toronto, Ontario

  - Programming in Java (specifically Servlets, Applets, Remote Method Invocation, JavaSever Pages, JavaBeans and JDBC), C++ and JavaScript.
  - Creation of Design Documentation using Unified Modeling Language in Rational Rose 2000 Professional J edition.
  - Managing Oracle SQL Server Databases on Windows NT and Sun Solaris and Microsoft SQL Server 7.0 on Windows NT.
  - Attended Oracle's "Managing Text and Multimedia Content with Oracle 8I" Seminar.
  - Participation in code reviews.
  - Increased exposure to UNIX and commands accessible exclusively to the super user.
  - Configuration of Netscape's Lightweight Directory Access Protocol between Netscape's Enterprise Server and Directory Server.
  - Creation of HTML (Hypertext Markup Language) and XML (Extensible Markup Language) documents.
  - Usage of INTERSOLV PVCS Version Manager for project source control.

Education

- **M.Sc. Computer Science**, December 2003 (Expected)
  Queen's University, Kingston, Ontario, Canada
  Supervisor: Dr. James R. Cordy and Dr. Thomas R. Dean

  Research Focus: The different methodologies used to model/verify software using sanity tests, formal methods, and modelling/verification software. Part of my implementation involved the completion of a program that automates the transformation from Java to C++.

  Dissertation Title: Concurrency Analysis of Java RMI Using Source Transformation and Verisoft

- **B.Sc.(Honours) Computing and Information Science**, 2001
  Queen's University, Kingston, Ontario, Canada

Research Experience

- **Research Assistant**, September 2002 to Present
  Source Transformation Group, Software Technology Laboratory
  School of Computing, Queen's University
  Supervised by Dr. James R. Cordy and Thomas R. Dean

Teaching Experience

- **Teaching Assistant**, September to December 2002
  Department of Computing and Information Science, Queen's University
  Course: CISC 223 - Software Specifications

- **Teaching Assistant**, January to April 2003
  Department of Computing and Information Science, Queen's University
  Course: CISC 365 - Algorithms

Other Activities

- Actively involved in weight lifting and snowboarding.
- Avid reader of various types of literature.
- Volunteer at The Knox Presbyterian Church shelter.