Concurrency Analysis of Java RMI Using Source Transformation and Verisoft

Tim Cassidy Jim Cordy Tom Dean Juergen Dingel
Queen's University
School of Computing Science
Goodwin Hall
Kingston, Ontario
K7L 3N6
{cassidy, cordy, dean, dingel}@cs.queensu.ca

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 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 paper presents an approach 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. The paper discusses the details of the transformation and evaluates the approach on a number of small examples.

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[8]. 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 Verisoft.

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 is frequently used for the development of distributed networked 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.

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 which will always result in deadlock.

In Figure 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` 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 paper, we present an approach to analyze Java

```
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. Simple RMI code example that illustrates a Java RMI program that will definitely result in deadlock.

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. We use automated source transformation because it is less error prone and more scalable than a completely manual transformation. For instance, java.util.Hashtable and its dependent classes are composed of over 14,000 lines of Java code and were transformed into C++ code in under 10 seconds on a Pentium 4 2.20 GHz.

Our 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[4]. It is for this reason that partial order reduction has been shown to be an effective means to keep the state explosion problem in check.

1.1 Outline of Paper

The remainder of the paper is devoted to explaining how we transform Java code that makes use of RMI into C++ code that is analyzable by Verisoft. Section 2 addresses the various tools and libraries that are being used in this paper. Section 3 details the transformation process and the advantages of using our tool. Java possesses a garbage collection algorithm that automatically frees memory that is no longer being used. To mimic this behavior a rudimentary garbage collection algorithm is developed for C++ and detailed in Section 4. Section 5 lists the experiments executed and the metrics that were used to gauge those experiments. Section 6 details the limitations that were encountered in the transformational process. Lastly, a discussion will follow on the potential extensions to this work in Section 7 and other related work in Section 8.

2 Background

2.1 Verisoft

Verisoft[4] is a tool that can be used to analyze concurrent C++ programs. Verisoft simply traverses the state space

of a program up to an arbitrary depth (as set by the user) until it finds a deadlock, divergence, livelock or some user defined assertions. The depth of any state space traversal is dictated by the presence of visible operations. Visible operations are any functions from the Verisoft library, which include, but are not limited to, message passing operations and non-deterministic choice points.

Verisoft has successfully been used on, for instance, Lucent Technologies CDMA call-processing library[1], for Heart-Beat Monitor debugging and unit testing[5], and a variety of other programs.

2.2 Source Transformation

There are a wide variety of transformation languages that can be used to transform one language into another. 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[3].

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.

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[13]. In this instance, the change will be from Java source code to C++ source code.

2.3 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 has the same interface as the remote object. The stub class methods contains 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 illustrates this process.

3 Our Solution: Bridging the Gap

Figure 3 shows the overall structure of our transformation. There are three basic steps to go from valid Java RMI to C++ running in 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. In addition this step also involves the generation of the stub class (which acts as a proxy for the remote object) 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.

3.1 The Source Transformations

3.1.1 Step 1

The first step of Figure 3 is fairly straightforward. It consists of a semantics preserving transformation from Java source code to C++ source code. However, there are Java class methods that are declared `native` and thus have been written in a language other than Java (typically C or C++). In those instances, the methods must be written manually as the C or C++ is not freely available. The remainder of the limitations can be found in Section 6.

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

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

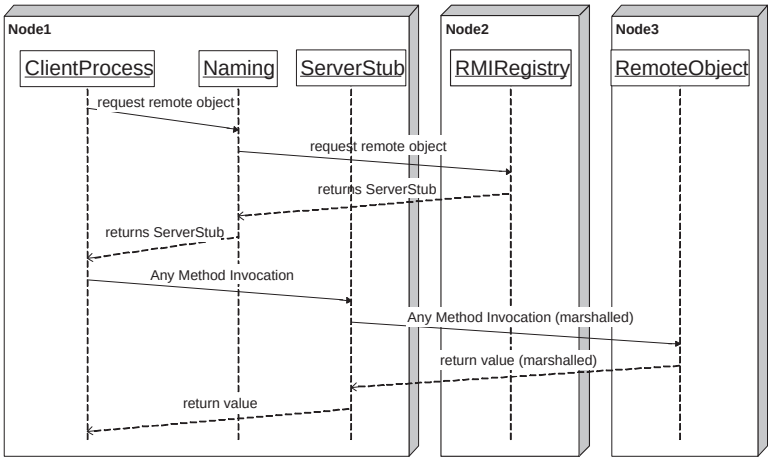

Figure 2. The sequence diagram for a Remote Method Invocation in Java RMI

java.lang.Object class. The name of this new class was `StdVector` and thus the transformation had to transform any Java array into a object of type `StdVector` in C++.

One of the transformational rules for the Java array to C++ `StdVector` class is illustrated in Figure 4.

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);
```

3.1.2 Step 2

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 isn't known until run-time. That is to say, to instantiate a class using a string.

Therefore, to model this behavior 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:

```
1  static SmtRemotePtr lookup (SmtStringPtr name_
       )
2  {
3    Object object = m_hashtable->get (name_);
4    if (object != '\0') {
5      if (instanceOf (object, Account)) {
6        return SmtAccountImpl_StubPtr (new
             AccountImpl_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.

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 it's contents actually send messages to the true remote object and wait 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.

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, the marshalling and unmarshalling of objects is currently not supported as outlined in Section 6.

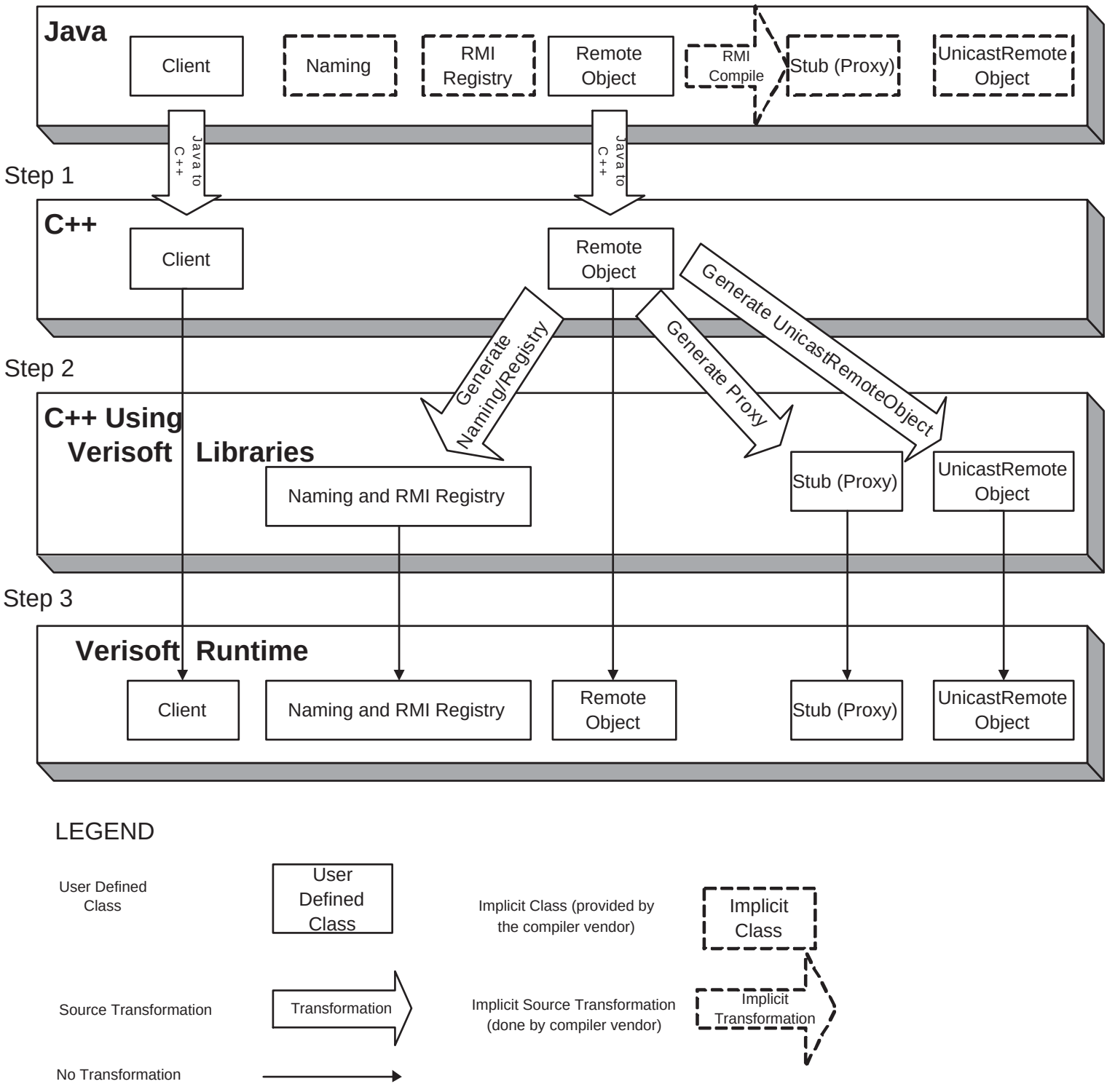

Figure 3. The sequence of transformations and subsequent execution of the model

```
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. A transformation from Java arrays to objects of type `StdVector`

3.1.3 Step 3

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

Then, Verisoft can be used in one of three modes (manual, guided, or automatic simulation mode) to analyze the resulting model. Manual mode is simply 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.

3.2 JTCUV Advantages

JTCUV increases the set of languages that Verisoft can analyze by performing an automated transform of Java code using RMI to C++ using Verisoft.

JTCUV is able to achieve a small reduction in the number of states in a Java RMI program, by extracting only the critical details of the message passing. The network aspect of RMI such as sockets and ports are not critical to our analysis of concurrency. Thus our resulting model will run on a single machine by making use of inter-process communication.

4 Implementation Details

In this section we outline the details of our solution and identify several particular challenges of transformation from Java to C++.

4.1 Memory Management

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

4.1.1 Memory Management in Java

Java possesses an advanced garbage collection algorithm which runs in the background while a Java program executes. Therefore, memory management in Java becomes very 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.

Our approach was to simply write a rudimentary garbage collection algorithm that involves smart pointers and reference counting.

4.1.2 Memory Management in C++

The stack is used by 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[9].

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[10].

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 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 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`).

Lastly, 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 behavior.

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 our approach to supplying a rudimentary garbage collection algorithm.

4.2 Two Garbage Collection Classes

Since there are a number of problems with memory management in C++ that don't 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.

4.2.1 Smart Pointer Class Implementation

The Smart Pointer class is provided so that your classes have all the benefits of being on the stack (i.e. will remove themselves from memory when they go out of scope) and all the benefits of being dynamically created (i.e. 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.

4.2.2 Base Class Implementation

The base class (in the C++ code written to support the transformation), named JTCUVobject, implementation 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 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 JTCUVobject to ensure that all classes extend JTCUVobject and thus allow for the self deletion used in the JTCUVobject class.

[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).

4.2.3 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 (JTCUVobject) 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 (JTCUVobject) and the following typedef is used to enhance readability:

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

5 Experiments/Results

The complexity of any of the transformations (in either stage 1 or 2) is O(n). The only exception is in the first stage there is one part of the transformation (specifically initialization lists in constructors) that causes the complexity to actually be O(n^2).

So far only two small RMI examples have successfully been transformed.

The aforementioned transformation of java.util.Hashtable and its dependent classes (from Section 1) was tested to ensure the behavior matched that of the original Java java.util.Hashtable. One hundred different C++ tests were written and executed successfully.

In the example provided in Figure 1 the transformation for the non-implicit classes (i.e. PeerA and PeerB classes) was completed within 5 seconds. The analysis by Verisoft (in which deadlock was found) also completed within 5 seconds.

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.

6 Limitations

6.1 General Limitations

Since the message passing modelling that Verisoft performs is inter-process communication, any message passing that occurs in Java across the network (i.e. using ports/sockets) is reduced to inter-process communication. Thus any networking problems or bugs in the Java RMI framework will not be found.

Also, like any modelling tool, as the model increases in size, the number of states increases exponentially. This is known as the *State Explosion Problem* and though it is reduced through the use of partial order reduction and by only monitoring visible operations, the size of the state space to be analyzed by Verisoft may still be prohibitively large as the size of the program increases.

Lastly, one of the disadvantages 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.

6.2 Limitations Specific to Our Implementation

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 our implementation (smart pointer in conjuncture 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.

6.2.1 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.

6.2.2 Unexecuted Deletions

Sun's Java Virtual Machine is not only able to handle self references but convoluted circular dependencies through the use of Sun's advanced Garbage Collection mechanism[12]. However, since the smart pointer class being used in the C++ code relies on the 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    //the member variable on the following line
         is a smart pointer object
5    DynTestPtr testPtr;
6
7    Test () :testPtr(this){}
8
9  };
```

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    //the member variable on the following line
         is a mundane pointer
5    Test * testPtr;
6
7    Test () :testPtr(this){}
8
9    ~Test(){
10     delete testPtr;
11   }
12 };
```

6.2.3 Other Limitations

The following are a list of the more trivial problems encountered in the transformation which were resolved with manual editing:

- Unique Naming/Renaming: Addition of new or modification of existing members in a class.
- String Usage: All operator overloading in C++ must be done on user defined types.
- Constructors: Calling constructors from other constructors of the same class.
- Accessing Static Members of a Class: Determination of static members across multiple Java classes is difficult because Java uses a period to access both static and non-static member variables.
- Name Hiding And Scope: When a member function in C++ is overloaded in a derived class, it hides the member function in the base class.
- Class Definitions and Circular Dependencies: If circular dependencies exist, code must be moved out of the header files and into the implementation files.
- Nested/Inner Classes: Java's inner classes are a specific type of nested classes, whereas C++ inner classes must have the functionality of Java's inner classes added to them manually.
- Static Virtual Members of Classes: These exist in Java, but not in C++.
- Marshalling and Unmarshalling of Objects: Parameters and returned objects cannot be marshalled or unmarshalled. This subsequently reduces the set of programs that can be transformed.
- Mapping From Transformed Code to Source Language: No provision for a mapping back to Java code from the transformed C++ code.

7 Future Work

Currently, there are a number of a limitations that are feasible to address in the near future while others are much more complex. The easiest limitation to address would be Class Definitions and Circular Dependencies while the most difficult would be creating a better garbage collection algorithm.

The essence of JTCUV 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 objects), 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 JTCUV on large and more realistic pieces of Java RMI code.

8 Related Work

8.1 Limitations of Other Modelling Tools

Most transformations that are done by other tools miss the underlying semantics of the language[7, 2].

An example of this is the transformation done by Bandera[2] in transforming Java to Promela. In this transformation important dynamic I/O functionality (sockets, files, etc) is impossible to transform. Though it is possible to model the behavior of these services (reading or writing from files, sending messages over ports, etc) 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 behavior of Java in the subsequent C++ program that uses Verisoft libraries.

Neither Bandera nor Java PathFinder are capable of transforming/translating Java RMI into a modelling language[14, 11]. The main problem lies in the abundance of native methods in the Java RMI framework.

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[7] which supports Linear Temporal Logic.

These analysis tools are well suited to examining similar aspects of programs using modelling languages that are more limited in their I/O capabilities than Java. In these instances, the behavior of the various I/O libraries must be modelled. That is to say, the essential properties (blocking, nonblocking, dependency relationships, etc) of the I/O functionality must be determined and then used in the particular modelling language.

However, there is a significant reduction in the state space of a program if the behavior 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, livelock, etc).

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 is significantly reduced.

9. Conclusions

Concurrency in any program can be a source of inconsistent problems which thus makes these problems very difficult to debug. Tools like Verisoft, Bandera[6] and Java PathFinder [7] are useful in detecting these problems. However, it is difficult and sometimes simply intractable to attempt a transformation from a programming language to a modelling language. JTCUV provides the user with an approach to complete this transformation. JTCUV is a three step transformation from Java code that makes use of RMI to C++ that uses Verisoft. Despite a number of limitations, we consider the work presented in this paper a very promising first step towards leveraging the benefits of Verisoft and TXL for concurrency analysis.

References

[1] 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.

[2] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, R. S. Laubach, and H. 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.

[3] J. R. Cordy, T. R. Dean, A. J. Malton, and K. A. Schneider. Source Transformation in Software Engineering Using the TXL Transformation System. *Journal of Information and Software Technology*, 44(13):827–837, October 2002. `http://www.cs.queensu.ca/home/cordy/Papers/SCAM01_TXLinSE.pdf`.

[4] P. 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.

[5] 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.

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

[7] 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.

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

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

[10] J. Liberty, V. Aklecha, S. Haines, S. Mitchell, A. Nickolov, C. Pace, M. Thakkar, M. J. Tobler, D. Xie, and S. Zagieboylo. *C++ Unleashed*. Sams, Indianapolis, Indiana, 1999.

[11] S. Stoller. Stony Brook University. Personal Communication, 2003.

[12] B. Venners. *Inside the Java Virtual Machine*. Artima Software, Inc., 2003.

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

[14] T. Wallentine. Kansas State University, Kansas. Personal Communication, 2003.