Available online at www.sciencedirect.com
SCIENCE DIRECT®
ELSEVIER
Electronic Notes in Theoretical Computer Science 141 (2005) 57–75
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Source Transformation for Concurrency Analysis

Timothy Cassidy, James R. Cordy,
Thomas R. Dean, Juergen Dingel[1,2]

School of Computing, Queen's University, Kingston, Canada

Abstract

Concurrent programming poses a unique set of problems for quality assurance. These difficulties include the complexities of deadlock, livelock and divergence, which can be extremely difficult to detect and debug. A variety of tools have been developed to assist designers and developers of concurrent applications. Some of these tools, such as VeriSoft, are specific to particular implementation languages, such as C++.
The Java Remote Method Invocation (Java RMI) package facilitates the implementation of concurrent applications, including those where processes reside on different hosts and communicate over networks. Unfortunately, it does not relieve the developer from the potential pitfalls of controlling concurrent access to remote objects, and may, in fact, make concurrency problems even more difficult to find.
This paper presents an approach that allows the VeriSoft state exploration system to be used to analyze Java RMI programs for deadlock, livelock, divergence, and assertion violations. The system works by transforming Java RMI programs into C++ programs where Java syntax, structure, concurrency and memory management are replaced by C++ equivalents and Java RMI communication has been transformed to VeriSoft C++ inter-process communication. We present the details of this transformation and discuss preliminary results for a number of small examples.

Keywords: source transformation, static analysis, concurrency

1 Introduction

Development of concurrent programs poses a unique set of problems for quality assurance. These difficulties include the complexities of deadlock, livelock and

[1] Email: tim_cassidy@tricolour.queensu.ca, (cordy,dean,dingel)@cs.queensu.ca
[2] This work is supported by the Natural Sciences and Engineering Research Council of Canada.

doi:10.1016/j.entcs.2005.05.012

divergence, which can be extremely difficult to detect and debug. As a result, a variety of static and dynamic analysis tools have been developed to assist designers and developers of concurrent applications. In particular, VeriSoft [10] provides a system for concurrency analysis of inter-process communication in C++ programs.

The Java Remote Method Invocation (Java RMI) package [24] facilitates the implementation of concurrent applications designed for networked environments, for example client-server systems, and is widely used for production applications in e-commerce and other distributed environments. Java RMI provides a higher level of abstraction by hiding low-level concurrency implementation details. However, it does not relieve the developer from the potential pitfalls of controlling concurrent access to remote objects, and in fact may make concurrency problems even more difficult to find.

While a number of tools have been developed to assist in the design and development of correct concurrent applications in Java, none of these have yet addressed Java RMI. This is in part because Java RMI's concurrency operations are hidden in an external library implemented using native methods, and in part because its concurrency semantics are at a higher level of abstraction, implicit in the methods of its library rather than directly represented using Java concurrency primitives.

Rather than retargeting existing analysis tools to understand Java RMI directly, which poses a number of difficult new analysis questions and would be at best an expensive and time-consuming process, in this paper we present an experiment in providing concurrency analysis of Java RMI programs using a different kind of solution: a semantics-preserving source to source transformation from Java programs using Java RMI to C++ programs using VeriSoft inter-process communication.

2 Motivation

Concurrent programming must deal with difficulties that are not present in sequential programs. Three of these problems are deadlock, livelock and divergence. A *deadlock* is a situation where two or more processes are each blocked awaiting resources or results from the others such that none of them can proceed. A *livelock* occurs when two processes/threads are able to change their state (i.e. they are not blocked) but are unable make any useful progress. And *divergence* occurs whenever no communication occurs between two threads or processes after a given period of time. Any message protocol that uses synchronization can be vulnerable to deadlock. More commonly, these protocols are vulnerable to divergences. In multi-user environments, divergences can

```
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();
        }
    }
}
```

Fig. 1. Simple example of a Java RMI program that will always deadlock.

make an application completely unusable as the number of users increases. Thus it becomes important to prevent both divergences and deadlocks.

A variety of tools have been developed to assist in the design and development of correct concurrent applications [4,15,21]. Some of these are based on abstract models from which code can be generated (in whole or in part), others are based on analyzing the source code. One of the latest tools is the VeriSoft state space exploration system [10]. VeriSoft provides a set of libraries and an execution engine that explores the state space of a concurrent program implemented in C++ using inter-process communication. In addition to deadlock, livelock and divergence, the developer can specify invariant assertions that must hold throughout the execution of the program.

Java remote method invocation (Java RMI) is commonly used for the development of modern distributed networked systems such as e-commerce applications. The appeal of Java RMI is that it frees the developer from having to worry about the details of network communication such as opening, connecting and closing 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 concurrent elements increases, because the developer may easily overlook the introduction of a circular dependency between remote objects that could result in deadlock. Consider, for example, the Java RMI code in Figure 1, which will always result in deadlock.

In Figure 1, `PeerB` is started first. Once it has bound itself to the RMI registry (using the `Naming.rebind` method), `PeerA` can be started. `PeerA` binds to the RMI registry (also using the `Naming.rebind` method) and then

enters its `run` method, locates `PeerB` using the RMI registry, and invokes `PeerB`'s `executeTask` method. `PeerB` then tries 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, resulting in deadlock.

VeriSoft is explicitly designed to analyze C++ programs using inter-process communication, and cannot be used with other languages and concurrency methods directly. The VeriSoft website describes one possible way of analyzing Java, which is to call the virtual machine from a C++ program. However, such a method treats the Java program as simply a black box, and severely limits the analysis that can be done on the Java program itself. In particular, it does not address the problem of representing the concurrency of Java RMI using VeriSoft inter-process communication, and thus does not allow for concurrency analysis of Java RMI. Moreover, Java RMI systems are by nature distributed, typically including several independent communicating Java programs, not just one.

As an experiment in bridging this gap, we have developed a different method for allowing Java RMI programs to be analyzed by VeriSoft: a set of formal source transformations from Java to C++ that preserves Java RMI concurrency behaviour using VeriSoft inter-process communication libraries. The resulting C++ program can be analyzed for deadlocks, divergences and livelocks using VeriSoft, and analysis results can be easily associated with the original Java source using method names. We use automated source transformation because it is less error prone than manual transformation, because it is (at least in theory) amenable to formal verification, and because it is easily scalable to handling practical size programs. For example, *java.util.Hashtable* and its dependent classes, which are composed of over 14,000 lines of Java code, are transformed into C++ code by our system in under 10 seconds on a 2.20 GHz Pentium PC with no possibility of clerical errors.

The main reason we have chosen to work with VeriSoft in this experiment is that it works directly with source code. Being able to perform the analysis directly on the source code has two advantages. First, the possibility of spurious analysis results due to modelling inaccuracies is reduced because no model needs to be constructed. Second, it is simpler to relate analysis output such as error traces or counterexamples back to the C++ source code, and by method name back to the Java source from which it was transformed. A third advantage of VeriSoft is the fact that it is *stateless*, that is, it performs its analysis without an explicit representation of the values of program variables. Stateless analyses are particularly well suited to the analysis of software systems with large amounts of complicated data such as e-commerce applications.

A final benefit of VeriSoft over other analysis tools is its use of *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 [10]. 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.

2.1 Outline of Paper

The remainder of this paper is devoted to explaining how we transform Java code that makes use of RMI into C++ code that is analyzable by VeriSoft. Section 3 introduces the various tools and libraries used in our work. Section 4 details the transformation process and the advantages of using our tool. Java uses a garbage collection algorithm that automatically frees memory that is no longer being used. To mimic this behaviour a simple garbage collection algorithm is developed for C++ and detailed in Section 5. An example of the final results of our transform is explained in Section 6. Section 7 gives a short overview of some of the other experiments we have performed. Finally, related work is described in Section 8, followed by our conclusions and future work in Section 9.

3 Background

3.1 VeriSoft

VeriSoft [10] is a tool that can be used to analyze concurrent C++ programs. It does this by traversing the state space of a program up to an arbitrary depth as set by the developer until it finds a deadlock, divergence, livelock or until some user defined assertion fails. 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 is a practical verification tool that has been used in industry to analyze concurrent applications such as the Lucent Technologies CDMA call-processing library [3] and to assist in the debugging and testing of mission-critical systems such as the 4ESS Heart-Beat Monitor [11].

3.2 Source Transformation

There are a wide variety of transformation systems that can be used to transform one language into another. TXL [8] was developed over ten years ago to be used as a tool for exploring programming language dialects. Since that

time, it has been used for a variety of source transformation applications ranging from simple syntactic replacements to sophisticated software engineering transformations [9].

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. The transformations are described by example, using a set of context-sensitive formal transformation rules from which an application strategy is automatically inferred.

The transformation we use here is a *source migration*, a translation of the program from one language to another language at essentially the same level of abstraction [20]. In this instance, the change is from Java source code using Java remote method invocation to C++ source code using VeriSoft inter-process communication.

The source transformation for this migration is particularly challenging for a number of reasons. Firstly, the memory, concurrency and communication models of the two languages are quite different, posing interesting problems for preserving semantics. Secondly, the transformation of concurrency behaviour we need is not simply a language translation, but rather an interpretation of the semantics of a concurrency library at a high level of abstraction (Java RMI), and its representation using a different semantic model and level of abstraction (VeriSoft inter-process communication). Finally, the transformation involves both of the recently identified "hard problems" of source transformation: local-to-global rules [6] and coupled transformations [16].

3.3 Java RMI

The basis of Java Remote Method Invocation is very simple. The idea is that a remote object should "register" itself with the RMI registry by providing a network-wide unique name. In a sense, the registry acts like an internet-accessible hashtable for remote services. Following registration, any Java object can query the registry for a reference to the remote object, using the unique name by which the remote object registered itself.

When an object queries the registry, it gets back a reference to a stub class object which has the same interface as the remote object but is in the local context. The methods of the stub class contain socket-based requests to the remote object (with parameters "marshalled") and blocking code that awaits the receipt of the return object (from the corresponding remote method). If the remote method has a `void` return type, the local object simply blocks until it receives an acknowledgment that the remote method has completed. Figure 2 illustrates the process.

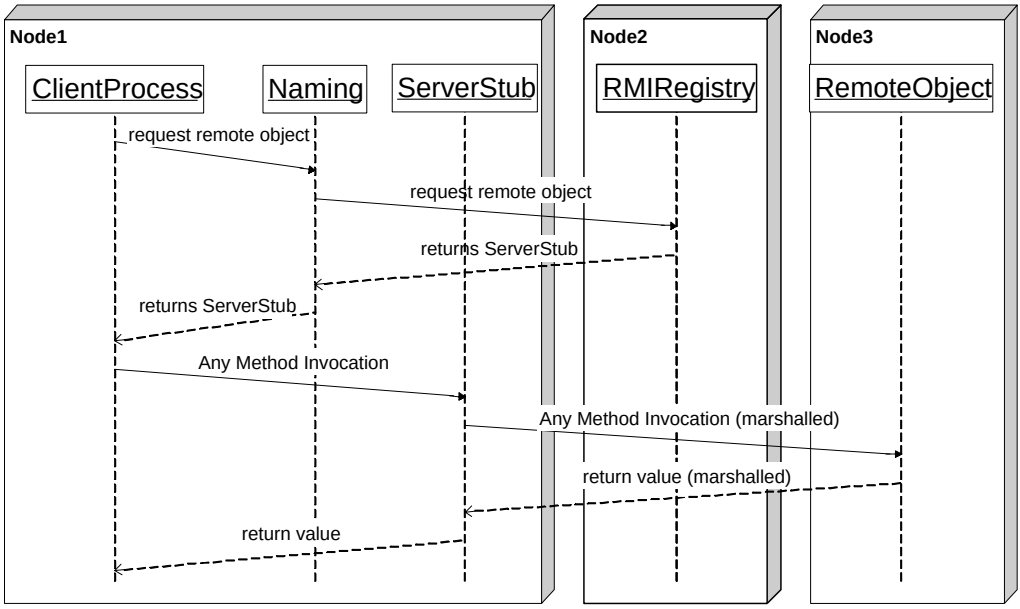

Fig. 2. The sequence diagram for a Remote Method Invocation in Java RMI.

4 Our Solution: Bridging the Gap

Figure 3 shows the overall structure of our transformation from Java RMI to C++ with VeriSoft. There are three basic steps. The first is an automated language transformation from Java to C++ using TXL. The resulting C++ code can then be compiled and executed once 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, including the generation of the stub class (which acts as a proxy for the remote object) and the *UnicastRemoteObject* class. This step uses TXL in a component generator role generating the required artifacts using source transformation.

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

4.1 The Source Transformations

4.1.1 Step 1: Language Migration from Java to C++

The first step of Figure 3, while large and complicated, is relatively straightforward. It consists of a semantics-preserving transformation from Java source code to C++ source code. In some systems there may also be Java class methods that are declared `native` and thus have been written in a language other

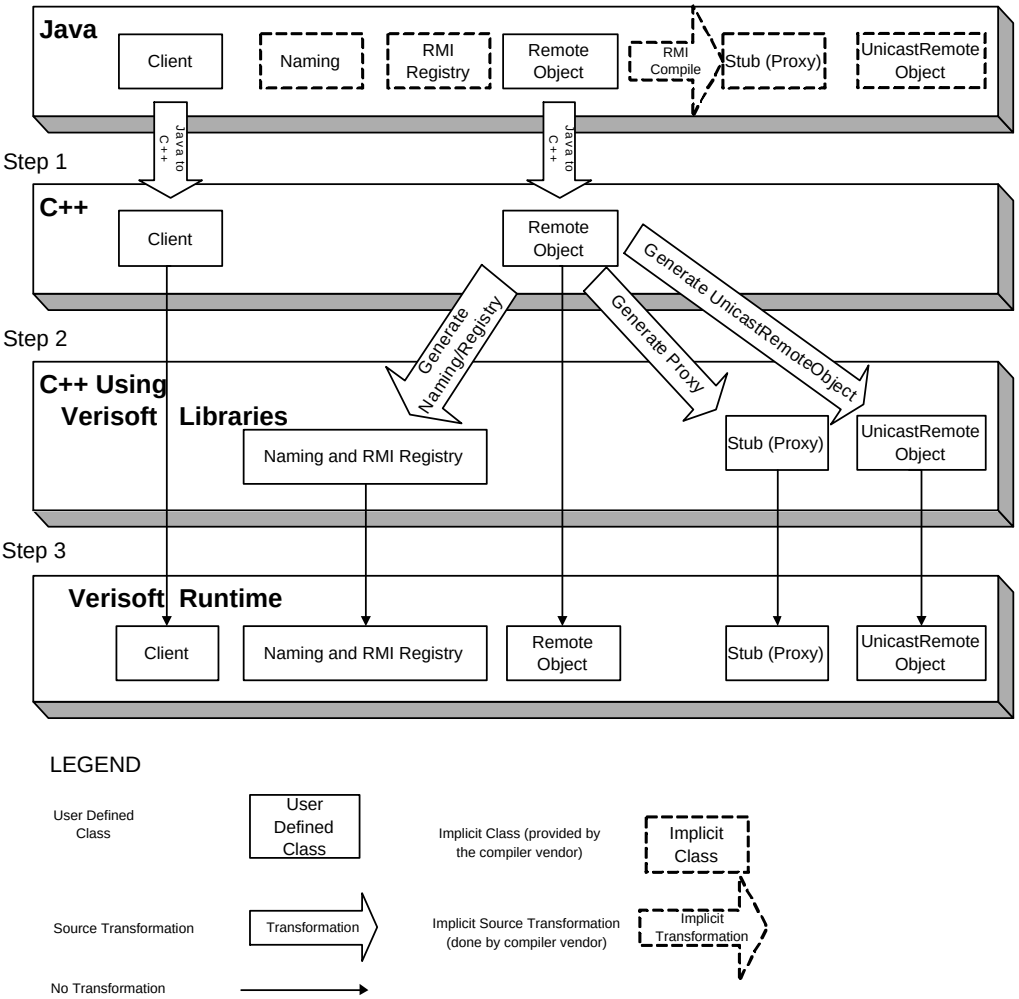

Fig. 3. The sequence of transformations and subsequent execution of the model. Implicit aspects of Java RMI must be generated as part of the transformation to C++.

than Java (typically C or C++). In such instances these methods may have to be handled manually since the C or C++ source code is not always available.

A simple example of the kind of transformation being done at this stage is the transformation of arrays. Although arrays are a simple construct, Java's arrays actually extend *java.lang.Object*, so the following is valid Java code:

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

Thus there is a requirement that the transformed *java.lang.Object* class and arrays (in C++) support this kind of assignment. To support this, a class was

```
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
```

Fig. 4. A TXL rule to transform Java arrays to C++ objects of type `StdVector`

written which essentially acts as a wrapper class (a class that provides access to the services of another class through its own methods) around C++'s standard vector class which also extends the transformed *java.lang.Object* class. The name of this new class is `StdVector`. In order to maintain Java semantics, the transformation must recognize and transform all Java arrays into objects of type `StdVector` in C++.

One of the transformation rules for converting Java arrays to C++ `StdVector` class objects is illustrated in Figure 4. Using it, the Java array assignment example above is automatically transformed into this C++ code:

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

4.1.2 Step 2: Generation of the Remote Object Registry in C++

The first part of the second step involves the generation of the *Naming* class, which implements a model of the Java RMI registry. This is required because C++ does not support any functionality similar to Java's reflection library, which provides (among other things) the ability to instantiate a class whose name is not known until run-time. In this case, we need the ability to instantiate a class using a string name.

To model this behaviour in C++ a *lookup* method is generated in the *Naming* class, which returns a remote class's stub based on its string name passed as argument. Here is an example of a generated lookup method in the *Naming* class generated for the example program of Figure 1:

```
1 static SmtRemotePtr lookup (SmtStringPtr name_)
2 {
3   Object object = m_hashtable -> get (name_);
4   if (object != NULL) {
5     if (instanceOf (object, PeerBInterface)) {
6       return SmtPeerB_StubPtr (new PeerB_Stub);
```

```
      }
    }
    return NULL;
}
```

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 with "`_Stub`" appended to it. If no such object exists in the hashtable, then `NULL` is returned.

Each stub class (proxy for a 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 wait for an acknowledgment (i.e. the methods are blocking). This requires replacing all network communication in Java RMI with VeriSoft methods that make use of inter-process communication.

The generation of stub classes is implemented as a source transformation of a copy of the transformed C++ version of the original Java RMI remote object class (Figure 5). For example, the following stub method is generated as part of the stub class for the remote class `PeerB` of the example program of Figure 1:

```
virtual void executeTask ()
{
  char * message = new char [100];
  sprintf (message, GLOBAL_executeTask_VAR);
  send_to_queue (m_remoteObjectMsgQueueID, QSZ, message);
  delete[] message;
  message = (char *) rcv_from_queue (m_msgQueueID, QSZ);
  if (strcmp ( message, GLOBAL_executeTask_VAR) == 0) {
    // Received the ack I was waiting for - remote method
       completed
    return;
  }
  throw SmtRemoteExceptionPtr ("problem in transmission of
     message");
}
```

This stub method begins by putting the string of characters represented in the macro `GLOBAL_executeTask_VAR` into the `message` character array. Then it sends those characters to the VeriSoft messaging queue variable named `m_remoteObjectMsgQueueID`. The messaging queue object is a VeriSoft-specific class used to send messages between processes. Once those characters have been sent to the queue, the buffer containing them is deleted locally. The code then blocks until it receives a return message from the queue. If the received message is what it expects (i.e. the same message it sent) then it returns, otherwise the smart-pointer version of a `RemoteException` object is thrown.

```
rule defineFunctions
    % make stub versions of all the methods in the class
    replace $ [member]
        % begin with the C++ version of an original method
        OptAccessSpecColon [opt access_specifier_colon]
        DeclSpecifiers [decl_specifiers] PointerOperators [repeat pointer_operator]
        FunctionNameID [id] DeclaratorExtensions [repeat declarator_extension+]
            OptCtorInit [opt ctor_initializer]
            OptExceptionSpec [opt exception_specification]
        FunctionBody [function_body]
    % make sure the method is not one of our generated ones
    deconstruct not FunctionNameID
        'AddRef
    deconstruct not FunctionNameID
        'Release
    % construct the global macro name
    construct MacroIDFromFunctionNameID [id]
        FunctionNameID [createMacroIDFromID]
    % now replace the original with a stub version communicating
    %     with the real thing using interprocess communication
    by
        OptAccessSpecColon DeclSpecifiers PointerOperators
        FunctionNameID DeclaratorExtensions
        {
            'char * message = new char '[100 '];
            sprintf (message, MacroIDFromFunctionNameID);
            send_to_queue (m_remoteObjectMsgQueueID, QSZ, message);
            'delete '[ '] message;
            message = (char *) rcv_from_queue (m_msgQueueID, QSZ);
            'if (strcmp (message, MacroIDFromFunctionNameID) == 0) {
                '// Received the ack I was waiting for - remote method completed
                'return;
            }
            'throw SmtRemoteExceptionPtr ("problem in transmission of message");
        }
end rule
```

Fig. 5. A TXL rule to transform every method of a C++ transformed Java RMI class into a stub method that communicates with the corresponding method of the remote object using VeriSoft inter-process communication

All remote objects in the Java RMI framework must extend the class *UnicastRemoteObject*. The generation of *UnicastRemoteObject* is necessary in order 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.

Like the stub classes, the `UnicastRemoteObject` class is generated using a source transformation based on a copy of the transformed C++ code for the original Java RMI remote classes. The TXL function implementing the

```
function createRunMethod  Members [repeat member]
  % generate the main run method given all the stub members
  replace [repeat member]
    % (generating, so nothing to begin with)
  % make selector if statements for each of the remote methods
  construct SelectionStatements [repeat statement]
    _ [createConditionals Members]
  construct FirstStatementAsRepeat [repeat statement]
    'message = (char *) rcv_from_queue (m_msgQueueID, QSZ);
  by
    'public:
      'void run ( )
      {
        'char * message;
        'while (1) {
          FirstStatementAsRepeat [. SelectionStatements]
        }
      }
end function
```

Fig. 6. A TXL function to generate the `run` method of the `UnicastRemoteObject` class for a transformed Java RMI program given as parameter the C++ transformed remote methods of the program

generation of the `run` method part of this class is shown in Figure 6. Here is the `run` method generated by this function as part of the `UnicastRemoteObject` for the program of Figure 1:

```
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 }
```

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 to represent the object so that it can be sent over a network, and unmarshalling is the process of reconstructing the object from the byte array received at the other end. Marshalling and unmarshalling of objects is supported by our current implementation only for simple integers and strings. However, the implementation can relatively easily be extended to handle any serializable object as a VeriSoft string message parameter.

4.1.3 Step 3: Compile, link and execute in VeriSoft

The last step simply involves compiling and linking the transformed and generated C++ files and executing the result using VeriSoft. Before executing, a VeriSoft `system_file.VS` file must be configured for the program. Factors such as the number of processes that will execute, the analysis depth (i.e. how deep in the state space to explore), whether to ignore deadlocks, and so on must be specified in this file in order for VeriSoft to run.

VeriSoft can then be used in one of three modes (*manual, guided,* or *automatic* simulation mode) to analyze the resulting model. Manual mode allows the user to 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.

5 Memory Management

Memory management is a particular challenge for our implementation because of Java's built-in automatic heap recovery. This is implemented in Java virtual machines by a garbage collector that runs in the background as a separate thread while the Java program executes. C++, however, has no such built-in garbage collection and by default uses explicit *new()* and *delete()* calls to manage the heap. While garbage collection libraries exist for C++ [2,23], they place rules on the allocation and use of memory which may not exactly match the Java model or may be too complex for VeriSoft analysis. In order to address this issue, we decided to use a simple garbage collection algorithm that involves smart pointers and reference counting in order to keep the transformation simple and the VeriSoft state space small. However, our approach can easily be changed to use other memory management libraries.

Our memory solution introduces two new Java classes. The first is a *smart pointer* class that makes use of templates. This class is used in place of regular C++ pointers. It allows dynamically created C++ objects the advantage of staying on the stack (like automatic objects). As these pointers are copied and assigned, they adjust the reference count of the objects that they point to. Smart pointers have deconstructors to make sure that reference counts are adjusted appropriately when a pointer variable goes out of scope.

As an example, consider the following declaration and initialization:

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

Our transformation to C++ introduces a new smart pointer type for the pointer, then uses the type to declare the variable and initialize it:

```
1 typedef SmartPtr<TrainCar> SmartTrainCarPtr;
2 SmartTrainCarPtr trainCar(new TrainCar(42));
```

In order to facilitate the tracking of references that are made to an object, we designed another class (named *JTCUVobject*) that all other translated classes extend. This class plays the role of Java's *java.lang.Object* class in our generated C++ code. Thus our transformation builds for the generated C++ code the same inheritance hierarchy as the original Java program. All Java classes extend *java.lang.Object* directly or indirectly, so the generated C++ parent class for transformed Java classes that do not specify a parent is our *JTCUVobject* class.

6 The Final Result

The final result of the entire transformation of the example Java RMI class of Figure 1 is shown in figure 7. The *AddRef* and *Release* methods added by the transformation are artifacts of our simple reference-counting memory management strategy discussed in the previous section.

Our system increases the set of languages that VeriSoft can analyze by automating the transformation of Java RMI code to C++ with VeriSoft libraries. In addition, our system is also able to achieve a small reduction in the number of states in a Java RMI program, by extracting only the critical high level details of the message passing. Low level networking aspects of Java RMI, such as sockets and ports, are also not critical to our concurrency analysis and thus are abstracted away by our transformation. Our resulting model is therefore able to run on a single machine, making use of inter-process communication, rather than requiring a networked client-server environment.

There is no reason why the strategies and methods we have demonstrated in our transformation system for Java RMI could not be used for other languages and concurrency paradigms as well, increasing the range of applications that can be analyzed using VeriSoft even further.

7 Experiments/Results

Thus far we have only used our transformations on a small set of Java RMI examples with intentionally added concurrency errors for the purposes of anal-

```
#ifndef PeerA_H
#define PeerA_H

class PeerA;
typedef SmartPtr <PeerA> SmtPeerAPtr;
class PeerA : public UnicastRemoteObject,
              public PeerAInterface,
              public Serializable {

  public:
    virtual void AddRef () {
        UnicastRemoteObject::AddRef();
    }

  public:
    virtual void Release () {
        UnicastRemoteObject::Release();
    }

  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();
        }
    }

  public:
    static void main
      (StdVector<SmtStringPtr>::type 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 {

  public:
    virtual void AddRef () {
        UnicastRemoteObject::AddRef();
    }

  public:
    virtual void Release () {
        UnicastRemoteObject::Release();
    }

  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
```

Fig. 7. Simple example Java RMI program of Figure 1 transformed to C++.

ysis. The transformation of *java.util.Hashtable* and its dependent classes (from Section 2) was tested using one hundred different C++ test functions to ensure the behaviour matched that of the original Java.

Although as yet untuned, our system is already reasonably efficient. In the example provided in Figure 1 the transformation of the non-implicit classes (i.e. *PeerA* and *PeerB*) was completed within 5 seconds on a small PC. The analysis by VeriSoft (in which the deadlock was found) also completed within 5 seconds on the same machine.

Another of our examples is a simplified version of a networked financial transaction system. Clients in the system all share the same account, depositing money into the account and getting the balance. This example made use of 10 separate clients making remote method invocations on a shared 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, and the analysis by VeriSoft took about 10 seconds.

8 Related Work

Many transformations done by other tools actually miss the underlying semantics of the language [14,7]. An example of this is the transformation done by Bandera [7] in transforming Java to Promela. In this transformation important dynamic I/O functionality (sockets, files, etc.) are impossible to transform, though it is possible to model the behaviour of these services (reading or writing from files, sending messages over ports, etc.) by indicating whether methods are blocking or non-blocking, dependency relationships, etc.

However, because C++ is a language that (at least in theory) is capable of any I/O activity that Java is capable of, it should be possible to exactly emulate the dynamic I/O behaviour of Java in the subsequent C++ program that uses VeriSoft libraries.

Although both can model Java programs, neither Bandera nor Java PathFinder are capable of transforming/translating Java RMI into a modelling language [22,19]. The main problem lies in the abundance of native methods in the Java RMI framework. By automatically transforming to C++ at a higher level of abstraction our work avoids this problem.

Bandera's greatest advantage over VeriSoft, and hence our work, is its ability 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 [14] which supports Linear Temporal Logic.

These analysis tools are well suited to examining aspects of programs similar to those we address, but using modelling languages that are more limited in their I/O capabilities than Java. The behaviour of I/O libraries must be modelled by determining the essential properties (blocking, non-blocking, dependency relationships, etc.) of the I/O functionality and encoding them in the modelling language.

However, there is a significant reduction in the state space of a program if the behaviour is modelled in this way. Although these tools must model any dynamic I/O libraries by hand, 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.). Thus although tools like Bandera and Java Pathfinder suffer from the requirement to manually build models of most low level Java I/O libraries, they make up for the effort in that the subsequent

state space in their programs is significantly reduced.

Our system is not the only one to utilize source transformation to assist in analysis of concurrent programs. A different kind of source transformation-based approach has been explored at Microsoft Research [18]. In the KISS ("Keep It Simple and Sequential") project, the idea is to transform concurrent programs to pure sequential programs augmented with instrumentation code to simulate execution of a large subset of the program's concurrent behaviour. In this way sequential program model checkers such as SLAM [1] can be utilized to analyze some aspects of concurrent behaviour, such as race conditions in device drivers. While this work does not cross language boundaries or free the user from the modelling task, it is similar to ours in two ways: it uses source transformation to extend the capabilities of an existing model checker, and it transforms a higher level abstraction of concurrency to a lower level representation with a different semantic model.

Another transformational method for analyzing concurrent Java programs has been proposed by Gradara et al. [12], who describe rules for transforming a limited subset of programs using native Java multithreading into equivalent CCS [17] specifications that can then be analyzed using temporal logic-based verification tools such as the NCSU Concurrency Workbench [5]. This work is particularly strong in its use of reduced models to address the state explosion problem. It is similar to ours in that the model is derived directly from the source using formal transformations and in its potential for automation. It differs in its use of an explicit intermediate model and in its focus on fine-grained mutlithreading rather than distributed remote concurrency.

9 Conclusions and Future Work

Concurrency in any program can be a significant source of quality assurance problems. Tools like VeriSoft [10], Bandera [13] and Java PathFinder [14] are useful in detecting these problems. However, it is difficult and sometimes simply intractable to attempt the hand transformation from a real programming language to a modelling language. In this work we provide the user with an alternative approach to solving these problems. We have developed a three step automated transformation from Java RMI to C++ using VeriSoft. It is a promising first step towards leveraging the benefits of VeriSoft and TXL source transformation for concurrency analysis of other languages and systems.

Since the message passing model that VeriSoft uses is inter-process communication, any message passing in the original Java across the network is reduced to inter-process communication by our transformation. Thus networking problems or bugs in message passing itself cannot be found using our

transformation. Another disadvantage is that VeriSoft does not support dynamic process creation, thus the tester must know the number of processes to be created at compile-time in order to do the analysis. This allows us to model and analyze Java RMI applications with any given number of clients, but it limits some of the options for the analyst. For bounded numbers of dynamic processes, this problem can be partially addressed by modifying the transformation to handle dynamic process creation automatically by mapping to a pre-allocated fixed number of static processes.

The most obvious need in this work is for continued testing on larger examples and production systems. The scalability of our solution faces three challenges: the sheer number of Java library classes on which a Java RMI program may depend, the ability of the TXL transformation to handle such large sources, and the ability of VeriSoft to analyze the large size of the resulting transformed C++ programs. Since both TXL and VeriSoft have already been proven to handle very large production programs in practice, it is actually the first of these that is the biggest threat to our technique.

Because we are limited to handling programs for which all used libraries have been transformed, every class on which a Java RMI program depends, directly or indirectly, must be transformed. It is the nature of Java that such sets can be surprisingly large; even relatively small programs may depend on scores of Java library classes, and thus the total size of the transformed C++ code may be large, potentially taxing the limits of the VeriSoft analyzer. A possible solution to this may be using VeriSoft's ability to use the Java Native Interface (JNI) to avoid transforming Java library classes that are not directly involved with Java RMI concurrency.

References

[1] T. Ball and S. K. Rajamani. The SLAM Project: Debugging System Software via Static Analysis. In *Proc. POPL 2002: ACM SIGPLAN-SIGACT 2002 Symposium on Principles of Programming Languages*, pages 1–3. ACM Press, 2002.

[2] H. Boehm. Dynamic Memory Allocation and Garbage Collection. *Computers in Physics*, 9(3):297–303, May/June 1995.

[3] 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.

[4] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *LNCS*, Copenhagen, Denmark, July 2002. Springer.

[5] R. Cleaveland and S. Sims. The ncsu concurrency workbench. In *Proc. CAV'96, 8th International Conference on Computer Aided Verification*, pages 394–397, July 1996.

[6] T. Cleenewerck and J. Brichau. An Invasive Composition System for Local-to-Global Transformations. In *Proc. LDTA 2005, ACM 5th International Workshop on Language Descriptions, Tools and Applications*, pages 44–63, April 2005.

[7] 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.

[8] J. Cordy. Txl – a language for programming language tools and applications. In *Proc. 4th Int. Work. on Language Descriptions, Tools and Applications*, pages 1–27, April 2004.

[9] 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.

[10] 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.

[11] 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.

[12] S. Gradara, A. Santone, M. L. Villani, and G. Vaglini. Model Checking Multithreaded Programs by Means of Reduced Models. In *Proc. LDTA 2004, ACM 4th International Workshop on Language Descriptions, Tools and Applications*, pages 55–74, April 2004.

[13] 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.

[14] 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.

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

[16] R. Lämmel. Coupled Software Transformations. In *Proc. SET 2004, First Int. Workshop on Software Evolution Transformations*, pages 31–35, November 2004.

[17] R. Milner. *Communication and Concurrency*. Prentice-Hall, New York, 1989.

[18] S. Qadeer and D. Wu. KISS: Keep it Simple and Sequential. In *Proc. PLDI 2004, ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 14–24, June 2004.

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

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

[21] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model Checking Programs. *Automated Software Engineering Journal*, 10(2), April 2003.

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

[23] P. Wilson. Uniprocessor Garbage Collection Techniques. *Proc. Int. Workshop on Memory Management*, (637), 1992.

[24] A. Wollrath, R. Riggs, and J. Waldo. A Distributed Object Model for the Java System. *USENIX Computing Systems*, 9(4):265–290, Fall 1996.