Transformations for Model Checking Distributed Java Programs *

Scott D. Stoller and Yanhong A. Liu

Computer Science Dept., SUNY at Stony Brook, Stony Brook, NY 11794-4400

Abstract. This paper describes three program transformations that extend the scope of model checkers for Java programs to include distributed programs, *i.e.*, multi-process programs. The transformations combine multiple processes into a single process, replace remote method invocations (RMIs) with local method invocations that simulate RMIs, and replace cryptographic operations with symbolic counterparts.

1 Introduction

There is growing interest in model checking of programs written in standard programming languages. Java's support for remote method invocation (RMI), an object-oriented version of remote procedure call (RPC) [BN84], makes writing distributed programs relatively painless. The current generation of model checkers for Java programs, *e.g.*, [BHPV00,PSSD00,Sto00,CDH$^+$00], work for multi-threaded single-process programs but not distributed programs. This paper describes three program transformations that extend the scope of these model checkers to include distributed programs, *i.e.*, programs that involve communication among multiple processes.

Centralization: merge all processes into a single process. This yields a non-distributed program.

RMI removal: replace RMIs with ordinary method invocations that simulate RMIs.

Pseudo-crypto: replace cryptographic operations with symbolic counterparts, which we call *pseudo-cryptographic* operations.

All three transformations improve performance of model checking. Centralization avoids the overhead of initializing multiple processes and Java Virtual Machines (JVMs). RMI removal replaces genuine RMIs with faster simulated RMIs. Pseudo-crypto replaces computationally expensive cryptographic operations, such as generation of public-private key pairs and verification of digital signatures, with symbolic counterparts that are at least an order of magnitude faster.

RMI removal and pseudo-crypto eliminate calls to native methods. Standard implementations of RMI invoke native methods for serialization (*i.e.*, conversion

* This work is supported in part by NSF under Grant CCR-9876058 and by ONR under grants N00014-99-1-0132, N00014-99-1-0358, and N00014-01-1-0109. Email: {stoller,liu}@cs.sunysb.edu Web: http://www.cs.sunysb.edu/~{stoller,liu}

of data into a format suitable for network transmission) and network communication. Cryptographic operations are typically implemented using native methods for efficiency. The simulated RMIs and pseudo-cryptographic operations introduced by RMI removal and pseudo-crypto, respectively, do not invoke native methods. This is significant, because supporting native methods in a model checker is a non-trivial task. Furthermore, native methods for network communication maintain state outside the JVM. Supporting such native methods in a model checker is extremely problematic, because there is no general way for the model checker to save and restore their state.

Centralization enables current model checkers for Java to be used to systematically test and verify distributed programs. It also suggests that there may be little incentive to extend those model checkers to directly handle distributed programs. Centralization can be used without RMI removal, because a single process may be both the client and the server in a RMI.

Centralization by itself does not attempt to control non-determinism from scheduling or other sources. Centralization is particularly useful in conjunction with tools that do provide some control. This includes debuggers as well as model checkers. For example, consider using the Java debugger jdb to debug a distributed program. jdb supports breakpoints, but a breakpoint halts only a single process. If jdb is used to debug the centralized program, then the breakpoint halts the entire system, which is often what the user wants.

Many distributed programs are designed to work over an insecure network, such as the Internet, and therefore use cryptography. During model checking, such programs are usually executed together with a program that simulates an adversary that controls communication over the insecure network. Cryptography causes a special problem for model checking, in addition to the issues of performance and native methods mentioned above. Specifically, if the program sends actual ciphertexts and does not help the adversary program determine their contents, then the adversary program would be extremely inefficient. For example, to determine whether it can decrypt an intercepted ciphertext, it would have to attempt the decryption with every key it knows. This problem arises in testing, as well as model checking. Pseudo-cryptographic operations enable the adversary program to efficiently determine the contents of intercepted ciphertexts.

All three transformations rely on the assumption that the original program is not real-time and does not use reflection in a way that would detect the transformation's effects.

2 Centralization

We refer to programs produced by the centralization transformation as *centralized programs*. The transformed program is equivalent to the original program in the sense that it has essentially the same possible executions as the original program. More precisely, (1) there exists a refinement mapping f such that, for each execution of the original program, there exists a stuttering-equivalent [Lam94] execution of the transformed program relative to f, and (2) *vice versa*.

The basic idea underlying centralization is simple. Suppose the original system consists of a process P_1 with three threads and a process P_2 with two threads. Then the centralized program creates five threads, three of which correspond to threads of P_1, and two of which correspond to threads of P_2. We assume that the original program does not count the total number of instances of `Thread` or `ThreadGroup` in the process, because this would detect the effect of the transformation.

Centralization involves four steps. The first step generates a driver class whose main method starts up the system. The driver is generated from a startup file supplied by the user, which contains (roughly) a list of command lines of the form "`java` $options_i$ $class_i$ $args_i$" used to start the processes of the original system. We refer to the process created by the i'th line of this file as process i. We assume that these processes can run on a single host, *i.e.*, different hostnames are not hardwired in the code. We currently do not support dynamic creation of processes, though it would not be difficult. The main method of the driver class creates, for each i, a thread that executes the main method of $class_i$ with arguments $args_i$.

The second step deals with static fields (*i.e.*, fields that are associated with a class rather than an instance of a class). In the original system, each process has its own copy of each class (the copy is created when the class is loaded by the JVM) and therefore its own copy of each static field. In the transformed system, there is only one copy of each class. The transformation introduces arrays to simulate the effect of having multiple copies of the class. For example, suppose the program uses a class C that contains a static field x of type T. The transformed version of class C declares a static field of type T`[]`, for an array whose elements have type T. Threads that correspond to threads of the i'th process access only C`.`x`[`i`]`. To allocate and initialize the arrays, we transform class initialization code—`<clinit>` and methods invoked directly or indirectly from `<clinit>`—appropriately.

Instructions that access static fields are transformed to access the appropriate element of the array. The index into the array is the number of the "process" to which the thread belongs. This value cannot easily be maintained in a global variable, because the JVM does not provide hooks that would invoke user-supplied code at every context switch. To determine this value efficiently, the transformation replaces all uses of `Thread` in non-library classes (*i.e.*, classes not in the Java 2 API) with `CentralizedThread`.[1] Class `CentralizedThread` extends (*i.e.*, inherits from) `Thread`, declares an instance field `int procNum`, and overrides the constructors of `Thread` with constructors that initialize `procNum` appropriately. When transforming an access to a static field, the index into the array is the value of the `procNum` field of the current thread; specifically, it is `((CentralizedThread)Thread.currentThread()).procNum`.

The third step deals with static synchronized methods, *i.e.*, methods that are associated with a class rather than an instance of a class and whose bod-

[1] Recall that, in Java, each thread is associated with an instance of class `Thread` or a subclass of `Thread`.

ies implicitly start with an acquire operation on the lock associated with the class and end with a release operation on that lock.[2] In the original system, each process has its own copy of each class and the associated lock. To simulate this, for each class C that declares static synchronized methods, the transformation introduces a static variable C`.locks` that points to an array of new objects, and, for each static synchronized method $C.m$, it inserts an acquire on C`.locks[((CentralizedThread)Thread.currentThread()).procNum]` at the beginning of $C.m$, inserts a matching release at the end of $C.m$, and marks $C.m$ as not synchronized.

The fourth step deals with the method `System.exit`, which terminates the process. In the transformed program, `System.exit` should terminate only threads with the same value of `procNum` as the invoker. Java does not directly provide a mechanism for one thread to terminate another thread. Transforming the program to incorporate such a mechanism is non-trivial. Currently, we transform calls to `System.exit` so that they throw `java.lang.ThreadDeath`, which should terminate the calling thread. This is correct if all other threads with the same value of `procNum` have terminated; this condition could easily be checked dynamically.

The current implementation does not transform static fields of library classes.

Centralization is independent of the communication mechanisms used in the program. It could be used in conjunction with a socket removal transformation as well as RMI removal.

3 RMI Removal

Java RMI works roughly as follows. A process, called a server, makes an object available to other processes, called clients, by registering the object in the RMI registry, which is a simple database that maps strings (names of services) to objects. A client locates a remote object by looking up a service name in the RMI registry. A successful lookup creates a new object o_{stub} in the client. o_{stub} is called a *stub* and contains the address of a server S and a reference to an object o in S. The stub is an instance of an automatically generated class, called a *stub class*. The stub class for class C is named C`_Stub`. A *remote reference* is a reference to a stub. For each remotely invokable method m of C, the automatically generated method C`_Stub`$.m$ on the client serializes its arguments *args* and sends them to server S; S unserializes the arguments, executes $o.m(args)$ in a new thread, serializes the return value (or exception), and sends it to o_{stub} on the client; the client unserializes the return value (or exception) and uses it as the result of the RMI. As an optimization, the JVM may maintain a pool of re-usable threads, rather than creating a new thread for each RMI. We do not describe here how interfaces are used to indicate which methods are remotely invokable; our transformation handles this aspect easily.

RMI removal replaces a RMI with an ordinary method invocation that simulates the RMI. The semantics of method invocation in Java is call-by-value

[2] Recall that, in Java, each class and each object implicitly contains a unique lock.

for primitive data, and call-by-reference for remote references and ordinary references. The semantics of RMI is different, because serialization followed by unserialization effectively performs copying. Specifically, the semantics of RMI is call-by-value for primitive data, call-by-reference for remote references (although the stub object is copied, the copy refers to the same remote object, not to a copy of the remote object), and call-by-deep-copy for local references. "Deep copy" means that the entire subgraph of the heap reachable from the arguments is copied; the copy is isomorphic to the original subgraph. In all cases, the semantics for passing return values is the same as for passing arguments.

The transformed program uses a simulated RMI registry. Currently, the simulated RMI registry expects to find stub classes in the `CLASSPATH`.

Which thread should execute a remote invocation? To ensure a faithful simulation of RMI, the transformed code could create a new thread to execute each RMI. This is easy to implement but inefficient and typically unnecessary, in the sense that most applications are insensitive to the identity of the thread that handles the RMI. Maintaining a pool of re-usable threads is not as easy to implement. In our current implementation, the calling thread executes the "remote" invocation; we assume that the application does not detect this difference. While the calling thread is executing a remote invocation of a method of an object o, the thread's `procNum` should be set to the number of the server that created o, because that is the number of the process on which the method would be executed in the original system. This requires an efficient mechanism for determining which process created each instance of each class with remotely-invokable methods. Accordingly, we insert a field `procNum` in each such class C and modify each constructor for C to initialize that field with the current thread's `procNum`.

Implementing copying using reflection is tempting, but reflection uses native methods, and we want to eliminate uses of native methods, so copying is implemented as follows. The transformation identifies classes whose instances might appear in arguments or return values of RMIs. For each such class C, it generates a method named C`.copyRMI`. Method C`.copyRMI` has a parameter `h` that indicates which objects have already been copied; it is used to ensure that the original subgraph and the copy are isomorphic. C`.copyRMI(h)` returns `this` if `this` is a remote reference,[3] and returns a deep copy of `this` if `this` is a local reference. In the latter case, C`.copyRMI(h)` starts by checking whether `this` is in the hash map `h`. If so, `this` has already been copied, so C`.copyRMI(h)` finds the copy using `h` and returns it. Otherwise, C`.copyRMI(h)` creates a new instance o of C, adds the mapping `this` $\mapsto o$ to `h`, copies from `this` to o the value of each field of C with primitive type, recursively invokes `copyRMI` on the value of each field of C with non-primitive type and stores the result in the corresponding field of o, and returns o. Creation and initialization of o require special treatment when C has a non-serializable superclass other than `java.lang.Object`; we omit details of how the transformation handles this.

[3] A more faithful alternative would be to clone (*i.e.*, create a shallow copy of) the stub, but the identity of a stub should not be significant to a normal application, so this cloning would be unnecessary overhead.

The transformation also generates stub classes. The standard stub classes produced by the compiler are not used by the transformed program, so we reuse their names for our stub classes. Thus, the stub class generated for class C is named C_Stub and declares an instance field target with type C, which refers to the object registered by the server. We still say that a reference to an instance of a stub class is a "remote reference". The generated method C_Stub.m($args$) creates new hashmaps hArgs and hRet, invokes copyRMI(hArgs) on arguments that are local references, sets the current thread's procNum to this.target.procNum, and invokes this.target.m on a combination of original arguments (that are not local references) and copies of arguments (that are local references); when the invocation of m returns, C_Stub.m($args$) restores the previous value of the current thread's procNum, invokes copyRMI(hRet) on the return value v of this.target.m if v is a local reference, and returns v (if v is not a local reference) or the copy of v (if v is a local reference) to the caller.

To efficiently check whether a reference is remote, the transformation introduces an interface StubInterface. All stub classes implement StubInterface. Thus, (r instanceof StubInterface) is true iff r is a remote reference.

Java RMI allows the user to specify a security policy that controls which remote methods may be invoked by which clients. Currently, we do not simulate checking of security policies; in effect, we assume that all RMIs performed by the original program are permitted by the security policy.

4 Pseudo-Cryptography

java.security provides a standard API for cryptography libraries. We assume that the original program uses this API. The original program is transformed by replacing all occurrences of java.security with PseudoCrypto; for example, java.security.Signature.sign is replaced with PseudoCrypto.Signature.sign.

The central issue in designing PseudoCrypto is the representation of ciphertexts. To solve the problem discussed in Section 1, the obvious approach is to use a symbolic representation. For example, the result of encrypting a plaintext t with a key k would be an object containing t and k. Given such a symbolic "ciphertext", the adversary program can trivially determine the key used to create it. This approach is standard in model checking of security protocols with traditional model checkers such as FDR and Murϕ [Low96,MMS97].

However, in the java.security API, ciphertexts are not an abstract data type. Ciphertexts are byte arrays, and this representation is visible to the application. Transforming the application to accommodate a different representation of ciphertexts would be difficult. Our approach is to maintain byte arrays and a symbolic representation. The symbolic representation is used only within PseudoCrypto and the adversary program; it is not visible to the application. A hash map is used to map the byte array representation of a ciphertext to its symbolic representation. A mapping is inserted in the hash map every time a ciphertext is created. The adversary program looks up intercepted ciphertexts in the hash map.

It is the responsibility of the author of the adversary program to simulate an adversary that controls an insecure network. For example, the author must determine which cryptographic keys stored in the hash table are known to the adversary.

If the original cryptographic operations are computed in the transformed program, then this transformation does not affect the contents of the byte arrays seen by the application and hence does not affect the behavior of the application.

Computing cryptographic operations during state-space exploration is often impractical, because of performance and native methods, as discussed in Section 1. Therefore, `PseudoCrypto` computes "pseudo-cryptographic" operations that maintain the byte array and symbolic representations, as above, except the byte arrays are filled with pseudo-random data, not genuine ciphertexts. This change in the contents of the byte arrays could affect the behavior of the application, *e.g.*, if the application does not trust its cryptography library and therefore checks whether ciphertexts have the expected format. Typical applications treat the byte arrays as "atomic values", merely passing them around and then using them as arguments to other cryptographic operations. Such applications are not affected by this transformation. Currently, manual inspection of the original program is used to check whether this could happen. A conservative automatic check based on static analysis could be developed.

Currently, we assume that all operations that produce ciphertexts involve pseudo-random initialization, so invoking an operation twice on the same arguments produces different ciphertexts. Operations that are "functional" (*i.e.*, whose return value depends only on the arguments) could easily be accommodated using memoization.

A separate issue is that the adversary program needs to efficiently determine which parts of intercepted "messages" might be ciphertexts. Currently, the author of the adversary program must deal with this. We are working on a static program analysis, based on points-to escape analysis [WR99], that aims to automate this and other some aspects of producing the adversary program.

Package `PseudoCrypto` currently supports commonly used methods for generation of public-private key pairs and generation and verification of digital signatures. Support for symmetric-key cryptography can easily be added.

5 Implementation and Case Study

The implementation of all three transformations is mostly complete. The implementation transforms bytecode using the Byte Code Engineering Library (BCEL), formerly called JavaClass [Dah99]. We hope to incorporate these transformations into Bandera [CDH$^+$00], which provides an excellent framework for model checking and the associated program analyses. The first public release of Bandera is expected soon.

The first major case study will be a secure and scalable distributed voting system, whose design is described in [MR98]. Students implemented that design in Java as a course project. We plan to model check it using the above transformations and Java PathFinder [BHPV00]. Java PathFinder is not yet available to us, but its first public release is imminent. We anticipate completion of this case study before SPIN 2001 and, if this paper is accepted, will present the results there.

Acknowledgments. Ziyi Zhou and Dezhuang Zhang implemented centralization. Srikant Sharma and Kartik Gopalan implemented RMI removal.

References

[BHPV00] Guillaume Brat, Klaus Havelund, Seung-Joon Park, and Willem Visser. Model checking programs. In *IEEE International Conference on Automated Software Engineering (ASE)*, September 2000.

[BN84] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2:39–59, February 1984.

[CDH$^+$00] James C. Corbett, Matthew Dwyer, John Hatcliff, Corina Pasareanu, Robby, Shawn Laubach, and Hongjun Zheng. Bandera: Extracting finite-state models from Java source code. In *Proc. 22nd International Conference on Software Engineering (ICSE)*, June 2000.

[Dah99] Markus Dahm. Byte code engineering with the JavaClass API. Technical Report B-17-98, Institut für Informatik, Freie Universität Berlin, 1999. Available via http://www.inf.fu-berlin.de/˜dahm/JavaClass/.

[Lam94] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[Low96] Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Proc. Workshop on Tools and Algorithms for The Construction and Analysis of Systems (TACAS)*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer-Verlag, 1996.

[MMS97] John C. Mitchell, Mark Mitchell, and Ulrich Stern. Automated analysis of cryptographic protocols using Murϕ. In *Proc. 18th IEEE Symposium on Research in Security and Privacy*, pages 141–153. IEEE Computer Society Press, 1997.

[MR98] Dahlia Malkhi and Michael Reiter. Secure and scalable replication in phalanx. In *17th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 51–60, October 1998.

[PSSD00] David Y.W. Park, Ulrich Stern, Jens U. Skakkebaek, and David L. Dill. Java model checking. In *Proc. First International Workshop on Automated Program Analysis, Testing, and Verification*, 2000.

[Sto00] Scott D. Stoller. Model-checking multi-threaded distributed Java programs. In *Proc. 7th Int'l. SPIN Workshop on Model Checking of Software*, volume 1885 of *Lecture Notes in Computer Science*, pages 224–243. Springer-Verlag, August 2000.

[WR99] John Whaley and Martin Rinard. Compositional pointer and escape analysis for Java programs. In *Proc. ACM Conference on Object-Oriented Systems, Languages and Applications (OOPSLA)*, pages 187–206. ACM Press, October 1999. Appeared in *ACM SIGPLAN Notices* 34(10).