# How to Migrate Agents*

Matthew Hohlfeld† Bennet Yee‡

August 14, 1998

## 1 Introduction

The idea of migrating processes is not new; implementations date back to the 1980s [20]. The early motivations for migrating processes were load balancing[1] and improved fault tolerance. Migrating autonomous agents is a more recent variation on this theme [10]. The goal of using agent architectures varies from intellectual property protection [1], fault tolerance [16, 18], scalability, to AI metaphors [19, 21]. This paper describes a novel mechanism for migrating programs in agent systems.

Previous systems such as LOCUS [20] provided process migration with full kernel support. To do this, these systems take a snapshot of the process's state, transfer it to another machine, and provide transparency for all system accesses. In contrast, agent systems are distributed yet localized: the agents explicitly migrate to access local resources, and so complete transparency is not important and perhaps inappropriate. While transparency may be a desirable feature for higher level abstractions such as automatic, transparent load-balancing systems, transparent migration is not an immediate, critical goal for agent systems. Furthermore, transparent migration can be built on top of a basic, explicit agent migration infrastructure.

Existing agent systems provide migration in one of two ways. Systems such as Telescript [8],[2] Agent Tcl [13], and ARA [15] provide migration by using a specialized language interpreter to capture the process's state. Other systems such as TACOMA [11], Mole [17], Odyssey [7], and Aglets [12] provide "migration" by a specialized remote execution mechanism: they finesse the migration issue by automatically migrating the code but not the current execution state. Unfortunately, this requires the agent programmer to explicitly write code to capture the relevant state variables and transfer their values in a message from one incarnation of the agent to the next. This makes writing agent code more difficult and error prone.

In this paper, we show how state capture — by recursively extracting the "local continuations" of all functions — can be achieved by simple source-level syntactic transformations of the agent's code. Similar transforms could be applied to object-level code to achieve the same goals. While our techniques are not restricted to Java and can be applied to programs written in other languages, Java will be our primary focus. Java's platform independence and incremental loading characteristics make it well suited for agent-oriented programming, and we are building an agent migration system based on applying our techniques to Java.

In Section 2 we provide a brief overview of the system, followed by examples of our transforms in Section 3. We discuss the properties of our transformation process in Section 4. In Section 5 we describe in detail our code transformations for Java, and in Section 6 we briefly discuss the agent server infrastructure requirements

*This research was funded in part by National Science Foundation CAREER Award CCR-9734243, a Faculty Development Award from National Semiconductor Corporation, and a gift from the Powell Foundation.

†hohlfeld@cs.ucsd.edu

‡bsy@cs.ucsd.edu

[1] For most job mixes, we now know that initial placement suffices for load balancing.

[2] General Magic's predecessor to Odyssey.

for supporting the use of our transforms. In Section 7, we examine how our code transformation techniques might be applied when using other programming languages, both for providing a migration "primitive" and for providing other language features, and discuss some immediate future work required to provide a complete mobile agent environment. In Section 8 we present our conclusions from this work.

## 2 Overview

Program migration systems vary in their expressive power. We first classify migration systems according to the migration abstraction that they provide to the application programmer:

1. *full migration.* All of the running program's state is migrated. This includes all threads' call stacks, kernel-mediated name spaces such as I/O descriptors, file-system names, etc., and other kernel-mediated state such as alarm timers, so that the migration is completely transparent. This is the type of migration provided by the LOCUS [20] operating system.

2. *thread-state migration.* The program's internal state is migrated, i.e., variables and call stacks. External name spaces, however, are allowed to change. This is the type of migration provided by systems like ARA [15], Agent Tcl [13], and Telescript [8].

3. *code-only migration.* The program is "restarted" at a remote site, possibly with initializing data in a message provided by the previous run of the program. The programmer is required to explicitly write code to record any partial results in the initialization messages so the new incarnation of the program does not have to repeat work. This is the type of migration provided by mobile agent systems like Odyssey [7], TACOMA [11], and Aglets [12].

Agent systems are arguably served most appropriately by thread-state migration, since explicitness of the execution location is often a goal of these systems. It is also beneficial that transparency with respect to the program's internal state makes writing agents easier. Code-only migration, on the other hand, provides only the minimal capabilities needed to implement agents. While this level of abstraction makes the agent system itself easier to implement, the agent programmers have to work much harder to write agents.

In this paper, we show that thread-state migration and code-only migration are equivalent.

Clearly, since operating systems and run-time systems for previous thread-state migration systems implemented their migration on top of simple communication channels, this is in a sense trivially true. What we show, however, is that thread-state migration can be provided via a pre-compiler, extending a base programming language like Java, similar to how C++ was initially implemented as a pre-compiler that generated C code as output. In particular, the standard base language compilers and interpreters, e.g., the Java bytecode compiler and Java Virtual Machine (JVM), can be used to process programs with our migration extension. This permits new advances in base-language compilers, interpreters, or just-in-time bytecode-to-native code compilers to be used in an orthogonal fashion.

Moreover, we present explicit, simple code transformations which allow agent programs to be written in a superset of Java, which has been extended by adding a `migrate` function. A pre-compiler transforms such programs into an equivalent program in Java that contains inserted code to perform thread-state migration. Our pre-compiler differ from other projects such as [2, 5, 9] that use Java or Java bytecodes as the target language of compilers in that we largely preserve the original code's structure — our transformations are purely syntactic.

Throughout this paper, our transformations will be described in terms of writing agents in the Java language. We refer to our techniques as the *migration* extension to the Java language.

Some features of the Java language such as object serialization (which first appeared in version 1.1 of the Java language) make the implementation of migration easier. In section 7, we will see what restrictions, if any, its absence imposes on the use of our techniques (or on programs that use our techniques) when used with other common programming languages.

Critical to our technique is the idea of allowing each method to return a *local continuation*, or data returned to the caller that allow the called method to be continued later where it had left off (i.e., lexically immediately after where it returned the local continuation). Unlike normal continuations, when invoked, a local continuation returns normally to the method that invoked the local continuation, rather than to the method that originally called the continuation-returning method.

While we describe the basic abstraction as an added `migrate` function, it is important to note that this is built on top of state capture and "release" operations implemented with local continuations. These basic mechanisms can also be used for implementing a generalized "remote fork" (`rfork`) facility, for process checkpointing, for explicit continuation-passing styles of programming, or for remotely spawning many copies of a program in parallel. In essence, the `migrate` function captures the state in a snapshot and sends the snapshot to the remote target machine. Support code at the remote machine receives the snapshot and releases it, reconstituting the program so that execution appears to continue immediately after the `migrate` function. This is how our system provides thread-state migration.

Our state capture and release primitives work recursively, one stack frame at a time. The automatic transforms required to implement capture and release involve modifying the original code so that additional processing occurs in each function prolog and epilog, and at every call-site. Examples of these transformations are presented below in the next section, and more details on the transformations are in Section 5.

# 3 State Capture and Release

By adding a `migrate` function to a base language, we provide the following abstraction: when a process executes the `migrate` function, the process's state is captured and sent to a target host machine, where it is *released*, i.e., another process is created such that its internal state is indistinguishable at the language level from the original. To simplify the exposition, we provide example code fragments in a slightly modified version of Java, where all method-local variable declarations are before any executable statements, and forward `goto` statements — which can jump into scope — are allowed. Note that such forward `goto` statements may be simulated by testing additional flags at every conditional branch and loop control expression to branch around code. To get to `catch` blocks, we re-throw the restored `Throwable` upon entering the corresponding `try` block.

## 3.1 Basic State Capture/Release

Capturing the state of a thread occurs recursively. Any procedure which can cause a migration as a side effect — including the `migrate` function itself — is modified by the pre-compiler to return a tagged union type. First, it may return "normally", in which case its original return value is available. Second, it may return "early", due to migration. In this case, the called procedure returns the information needed to reconstruct its state, including the state of any procedures that it called. This information is the local continuation for the procedure, which is used later to restart the function's execution at the point immediately after where it returned the continuation. Every call site of these procedures is modified by the pre-compiler to check the mode of the procedure's return: returning a local continuation causes the procedure containing the call site to capture its own state, and likewise return its own local continuation. Capturing the local state of an invocation of a procedure is done by copying all the local variables into a buffer — effectively a copy of the stack frame, which we call a *migration frame* — and a number indicating where in the body of the procedure a called function returned the migration request, effectively an abstract program counter value. This information is returned with the local continuation from the called procedure embedded. Note that since object references are pointers, our migration frames are analogous to the heap-allocated invocation frames of ML — they prevent the referred-to objects from being garbage collected. Unlike ML, instead of always allocating invocation frames on the heap, we switch from a stack to a heap only when a migration requires it.

Note that the `migrate` function is just a special case of the class of procedures that can cause a migration as a side effect. It has no local state other than the migration parameters. The `migrate` function will always return a local continuation when called with a null migration frame argument, so the actual calls are optimized away. The abstraction for programmers is that it acts as a procedure, but in the examples below we show the optimized transforms. (When `migrate` is called with a non-null migration frame, it is just returns the migration status.)

Because call sites are modified to return local continuations whenever the called method returns a local continuation, eventually, the state capture recursion bottoms out at the original `main` procedure. Here, the pre-compiler has renamed `main`, so it too can return "early" — to migration run-time support code. This code extracts the migration parameters to the `migrate` function, connects to the target host, and performs any authentication and key exchanges needed to establish a secure connection. Finally, it uses object serialization to transmit the transitive closure of all accessible objects, both globally and through the migration frames, to the remote host. As we will see in Section 7, while object serialization is a language feature not widely available in languages other than Java, it can be effectively added to other strongly typed languages. At the remote host, the serialized objects are reconstructed and a reverse process reconstructs the call stacks.

Let us look at an example of our transformation process. In our code fragments below, we demonstrate the migration transformations and the code need not make sense algorithmically. For expository purposes, we use easily remembered names for all of the classes, variables, functions, and labels that we introduce through our transformations. In practice, these names will effectively be in a separate namespace: we use a standard name prefix (which agent programmers should not use) that prevents collisions between our variables and those used by programmers in their agents.

Figure 1 shows an example code fragment which includes a `migrate` function. Figure 2 shows the transformed code. Note that the method `migratingMethod` now takes an additional argument and has a different return type. In a normal call, the added argument `capturedFrame` will be `null`, and the code's execution will be identical to the original code between the initial test of `capturedFrame` and the transformed `migrate` function. The transformation replaces the `migrate` function with code that saves all local variables — whether primitive types or object references — into the `captureReturn` object, and returns this object.

The returned object has the following important property: if it is passed into `migratingMethod` as its first argument, the transformed code will reset the local variables to the saved values and then unconditionally jump to the position immediately after the `migrate` call's location in the original code — this is how a local continuation is "invoked".

The returned local continuation object is invoked, of course, as part of the state release process. Passing a returned local continuation object as the continuation parameter to a function will result in a state identical to the time when the continuation was captured, so long as all of the ancestral callers' states are likewise identical. The state release continues recursively, until it bottoms out at where the `migrate` call was located in the original code.

All the callers of `migratingMethod` must also be automatically modified in order to work with the transformed version of `migratingMethod`. Figure 3 shows an example code fragment which invokes `migratingMethod` and Figure 4 shows the corresponding transformed code. Again, the example code is written to demonstrate the mobility transform and does not make sense algorithmically. Note that the variable `migrationLocation` is used to distinguish a normal return of the method from a return of a local continuation.

The state capture and release transformations require the creation of new classes — in our example,

```
class C {
        int cv1; /* class variables */
        retType migratingMethod(int a1,String a2 /* arglist */) {
                int i, j, k;
                retType retValue; /* ... var decls... */
                i = a1 + a2.length + cv1;
                retValue = new retType(...); /* others statements */
                k = migrate(targetHost);
                j = 5 * i + 2 * cv1; /* others statements */
                return retValue;
        }
};
```

Figure 1: State Capture at `migrate`: the original code

`capture_retType` and `capture_retType2` — to store the auxiliary data required by our state capture/release algorithm. These classes are automatically derived from the list of local variables in a function and the type of the function itself.

The cost of the basic transform is minimal. Assuming that the frequency of migration is extremely low compared to method invocations, the cost is basically an extra object creation per method invocation — which is immediately made inaccessible, an extra test per method return, and tests of flags (for simulating the forward `goto` statements) in conditionals which will almost always be false. While the object creation/destruction will incur extra garbage collector overhead, this is well handled by generational garbage collectors and may even be optimized away by smart compilers that perform interprocedural optimization.[3]

## 3.2 Migration and Throws

In the examples that we've given, the situation has been slightly oversimplified. This is because a `throw` can cause abrupt control flow transfers that could interfere with our migration transform, hiding state associated with the programs' control flow that is not captured by recording variables and call locations. Section 5.3 gives details on how this is handled. A detailed example of the transformations needed to handle exceptions is in Appendix A.

## 3.3 Migration and Threads

So far, we have seen how to migrate a single thread's call stack. To simplify the system design, we require that all threads in a multithreaded application migrate simultaneously. This prevents a single thread from migrating alone, obviating the need to implement distributed objects. Because of this limitation, transforming multithreaded applications requires a little more work. Since there is no way for one thread in Java to asynchronously force other threads to capture state, we have to set a global flag which is periodically checked by all threads in order for one thread (the one executing the `migrate` function) to make the other threads migrate with it. Because threads may have `synchronized` methods on their call stacks, threads cannot capture state independently — otherwise, the migratory, or continuation-capturing, return of a `synchronized` method can allow another thread to enter that method inappropriately.

[3]The dynamic incremental loading of Java classes make this more difficult in practice, since usage invariants discovered may be invalidated by newly loaded code.

```
class capture_retType implements Serializable {
        int migrateLocation, migrateStatus;
        Vector migrationFrames; int frameIndex;
        hostInfo targetHost;
        retType result; /* normal return */
        int a1; String a2; /* arglist */ int i, j, k; retType retValue; /* locals */
};
class C implements Serializable {
        int cv1; /* class variables */
        capture_retType migratingMethod(capture_retType capturedFrame, int a1,String a2) {
                int i, j, k; retType retValue; /* ... var decls... */
                capture_retType captureReturn;
                if (capturedFrame != null) {
                        int gotoLoc = capturedFrame.migrationLocation;
                        a1 = capturedFrame.a1; a2 = capturedFrame.a2;
                        i = capturedFrame.i; j = capturedFrame.j; k = capturedFrame.k;
                        retValue = capturedFrame.retValue;
                        capturedFrame.migrationFrames.setElementAt(null,0); /* allow gc */
                        switch (gotoLoc) {
                        case 1: k = capturedFrame.migrateStatus; /* k = migrate(targetHost) */
                                capturedFrame = null; /* allow gc */
                                goto CaptureLabel1;
                        }
                }
                i = a1 + a2.length + cv1;
                retValue = new retType(...); /* others statements */
                captureReturn = new capture_retType();
                captureReturn.a1 = a1; captureReturn.a2 = a2;
                captureReturn.i = i; captureReturn.j = j; captureReturn.k = k;
                captureReturn.retValue = retValue;
                captureReturn.migrateLocation = 1;
                captureReturn.targetHost = targetHost;
                captureReturn.migrationFrames = new Vector();
                captureReturn.frameIndex = 0;
                captureReturn.migrationFrames.setElementAt(captureReturn,0);
                return captureReturn;
CaptureLabel1:
                j = 5 * i + 2 * cv1; /* others statements */
                captureReturn = new capture_retType();
                captureReturn.result = retValue; captureReturn.migrateLocation = 0;
                return captureReturn;
        }
};
```

Figure 2: State Capture at `migrate`: the transformed code

```
class C2 {
        retType2 c2method(int b1, C b2) {
                int i;
                retType x;
                retType2 y;
                i = b1 + 5;
                x = b2.migratingMethod(b1,"Hello world");
                y = new retType2(i,x);
                return y;
        }
}
```

Figure 3: State Capture at Call Site: the original code

To prevent migration-induced synchronization errors, we introduce two barriers. First, after globally announcing migration or noticing a globally announced migration, a barrier is held until all threads are aware of the pending migration (e.g., using `sun.misc.Lock` and `sun.misc.ConditionLock`)[4]. Only after all threads have noticed the pending migration can state capture be initiated safely, since no thread will accidentally enter into a critical section then. When all of the threads' `Run` methods have captured their state, the thread states and global class variables may be sent via object serialization as before. When new threads are created and the thread states are released at the target host, a second barrier occurs at the apparent return from the `migrate` function or at the global-flag induced migration. By this, we ensure that all threads must also have had their states released, so that all `synchronized` methods will have been re-entered, and no thread's forward execution could accidentally have entered a `synchronized` method inappropriately.

The `ThreadDeath` exception, which is not normally caught by user code, will be caught by our transformed `main` function, so that it can keep track of the number of extant threads. With this addition and careful management of `thread` creation points, we can maintain a count of the number of threads in an agent.

In addition to inserting code to prevent synchronization errors, we need to insert code to keep track of threads about to invoke `synchronized` methods. We cannot simply allow threads to block awaiting entry into `synchronized` methods, since blocked threads cannot respond to a pending migration request, resulting in a migration-induced deadlock. We can avoid migration-induced synchronization errors simply by having all `synchronized` methods first check for the pending-migration flag and return a "migration retry" status flag. However, this strategy does not prevent deadlock. To prevent deadlock from occurring in the inserted code, each thread that will attempt to enter a `synchronized` method marks itself as such. The migration barrier code will not wait for it to notice the migration request, since the thread may be blocked waiting to enter a `synchronized` method already occupied by another thread. When the thread actually enters the `synchronized` method, it immediately checks the global pending-migration flag, and return "migration retry" appropriately.

Another type of overhead incurred by multithreaded agents occurs as a consequence of having a global pending-migration flag. Since we wish to make migration as transparent as possible, the pre-compiler needs to automatically insert code to periodically check the pending-migration flag. Choosing an appropriate frequency — or choosing a code location to achieve the appropriate frequency — is difficult: to limit the overhead, we want to avoid inserting extra tests into inner loops; to make migration actually occur in a bounded amount of time, we want to check the flag in every loop that might run for a long time. Exactly

[4] This implies that there is a global counter which keeps track of the number of extant threads, effectively `activeCount()` for all `threadGroups`.

```
class capture_retType2 implements Serializable {
        int migrateLocation; Vector migrationFrames; int frameIndex;
        retType2 result; /* normal return */
        int b1; C b2; /* arglist */ int i; retType x; retType2 y; /* locals */
};
class C2 implements Serializable {
        capture_retType2 c2method(capture_retType2 capturedFrame, int b1, C b2) {
                int i; retType x; retType2 y;
                capture_retType migrMethFrame = null;
                capture_retType2 captureReturn;
                if (capturedFrame != null) {
                        int gotoLoc = capturedFrame.migrationLocation;
                        b1 = capturedFrame.b1; b2 = capturedFrame.b2;
                        i = capturedFrame.i; x = capturedFrame.x; y = capturedFrame.y;
                        capturedFrame.migrationFrames.setElementAt(null,
                                capturedFrame.frameIndex);
                        switch (gotoLoc) {
                        case 1: migrMethFrame = capturedFrame.migrationFrames.elementAt(
                                        capturedFrame.frameIndex - 1);
                                capturedFrame = null; /* allow gc */
                                goto CaptureLabel2;
                        }
                }
                i = b1 + 5;
CaptureLabel2:
                migrMethFrame = b2.migratingMethod(migrMethFrame, b1,"Hello world");
                if (migrMethFrame.migrationLocation != 0) {
                        captureReturn = new capture_retType2();
                        captureReturn.b1 = b1; captureReturn.b2 = b2;
                        captureReturn.i = i; captureReturn.x = x; captureReturn.y = y;
                        captureReturn.migrateLocation = 1;
                        captureReturn.frameIndex = migrMethFrame.frameIndex + 1;
                        captureReturn.migrationFrames = migrMethFrame.migrationFrames;
                        captureReturn.migrationFrames.setElementAt(captureReturn,
                                captureReturn.frameIndex);
                        return captureReturn;
                } else {
                        x = migrMethFrame.result; migrMethFrame = null;
                }
                y = new retType2(i,x);
                captureReturn = new capture_retType2();
                captureReturn.result = y; captureReturn.migrationLocation = 0;
                return captureReturn;
        }
}
```

Figure 4: State Capture at Call Site: the transformed code

how to do this automatically is not yet clear.

It is apparent that handling multithreaded programs will incur a significant performance penalty, since the inserted code to periodically check the global pending-migration flag must execute even if migrations do not occur or occur infrequently. Unfortunately, we do not have any empirical measurements or experiments on "real life" agents from which we can tune this insertion procedure; this remains future work. Note, however, that this overhead is no different than that incurred whenever threads must be stopped in a clean way.

# 4 Our Migration Technique

The migration abstraction that we provide is thread-state migration. As we demonstrated in the previous section, it is easily implemented with a pre-compiler. In this section, we discuss the implications of using our migration technique.

## 4.1 Language Extension via Pre-compilers

Like the idea of migrating processes itself, language extension via pre-processing is an old technique. Instances of it range from macro assembly languages to early implementations of C++. Here, we examine our pre-compiler approach to providing a migration language extension and compared it with server-based approaches.

There are several benefits to our language extension approach. First, because the change to the language semantics by the added `migrate` construct is minimal, programmers already familiar with the base language should feel comfortable with writing programs in our extended language. Additionally, most standard tools for software development may be used, since the `migrate` construct is syntactically a simple method invocation. (Unfortunately, standard debuggers will not work with agents after they migrate.) Moreover, because the language extension can be achieved by a pre-compiler, advances in technology for the base language compiler implementations should apply to the extended language as well. For example, a new optimizing compiler or just-in-time bytecode-to-native-code compiler for Java will benefit not only standard Java code, but also agents written in Java with our migration extension.[5]

Our migration extension technique has some other security/software engineering implications as well. We were motivated to look at mobile agents because of the security problems inherent in this style of distributed computation [23]. By using a pre-compiler to implement migration, we reduce the amount of code in the trusted computing base. This has two important effects: the likelihood of an error in the trusted computing base is smaller, and upgrades of the migration mechanism do not affect the trusted computing base. Why the first property is desirable is obvious. The second property is also desirable for widely distributed systems: servers can be hard to upgrade rapidly, especially when they are widespread and under different administrative domains. The agent code, however, can be easily passed through a new pre-compiler. This obviates the need to upgrade the migration mechanism in every server that an agent might pass through.

There are also several drawbacks to our approach. Critically, the agent programmers must use only migration-aware libraries, or explicitly close references to local resources like I/O stream objects when migrating. For multithreaded code, functions from standard libraries must be non-blocking, or the time spent blocked must be bounded by a reasonable upper bound, since all threads must migrate simultaneously. While the language-level changes are minimal, the API-level changes are likely to be non-trivial. On the other hand, mobile agents are a relatively new class of applications, and restricting agent code to the use of

[5]Strictly speaking, this is not always true: the code transformations that our pre-compiler applies may make it more difficult for optimizing compilers to analyze the code and apply their optimizations.

specialized agent libraries is perhaps not entirely unreasonable.[6]

The above ease of upgrade argument is diametrically opposite to the arguments usually made in favor of centrally administered systems. According to that argument, it is easier to upgrade the central servers so that the benefits are immediately available to all user programs. The analogy to allowing centralized updates is implementing the migration primitive as a feature of the agent server, for example, as an extension to the JVM. In the pre-compiler approach, agent dispatchers will not reap any benefits unless they know that a new pre-compiler or optimizing base-language compiler is available and reprocess their agent code through the new tool(s). Furthermore, if the need for using a new code transformation tool is due to a bug which makes the pre-compiler generated agent code behave incorrectly, the pre-compiler approach would leave those dispatchers who have not upgraded vulnerable. While upgrading many distributed servers is difficult, presumably the number of such upgrades is fewer than that required to update the pre-compiler for all agent dispatchers. Arguably, most servers may be better administered than most client machines on the Internet, so getting all the agent servers to upgrade to a new JVM may not be so bad in practice. We believe, however, that the the pre-compiler approach results in fewer interdependencies and makes distributed administration much easier, and thus is the superior alternative.

## 4.2 Automatic Continuation Extraction

The `migrate` construct is implemented by inserting code to automatically extracting the current thread's continuation. We can think of `migrate` abstractly as composed of two steps. First, invoking an `extract-current-continuation` function (which is used much like the C standard library `setjmp` construct), sending the resultant continuation to the destination server, and exiting. Next, the receiving host "calls" the continuation, thereby rebuilding the thread's stack, releasing the captured state. Other abstractions such as `rfork` may be built on top of the `extract-current-continuation` primitive as well.

The returned local continuation behaves similarly to a normal continuation when the variables local to the function are object references. Variables of a primitive type such as `int`, however, behave differently: repeated invocations of the local continuation will reset the primitive variables to the values they had when the continuation was captured rather than allowing them to accumulate changes. Note that the pre-compiler could make the code always allocate a migration frame — just as ML does heap allocation of activation frames — and always use "local" variables from there, thereby making the behavior of local continuation for primitive types identical to that of normal continuations. Such strict adherence to the semantics of normal continuations, however, is unnecessary for our agent migration work. Furthermore, if such behavior is desirable, compilers for ML or other strongly typed languages with first class continuations could directly target the JVM as an alternative to making semantically small extensions to Java.

Our technique of automatically extracting the continuation, while heavy-weight, may have other applications. Programming techniques that are used in languages with continuations as first class objects can be adapted for languages without explicit continuations by the use of our pre-compiler technology.

## 4.3 Programmer Visible Abstraction

Using our migration extension technique, it is possible to create an agent system in which the programmer is not restricted from using any part of the normally available functionality in their programming language. As mentioned above, any library calls must be non-blocking when multithreading is used. We can handle most blocking methods with further code transformations, but we are as yet unconvinced that this is necessary: mobile agent applications probably need to use special, migration-aware libraries anyway.

[6]The exact design of the library support for agents written with our migration extension is as yet undetermined; we are presently concentrating our efforts on the implementation of the pre-compiler.

The migration request itself appears to the programmer as a simple function call. That is, we ensure that migration requests can occur at any point in the original source that a function call could normally occur. This request includes a destination and optional parameters for precisely controlling the migration. The `migrate` function returns a status value indicating success or the reason for failure.

Due to the lack of location transparency, agents will see changes in external name-space mappings as a result of a successful migration. Note that the state release algorithm only restores object values: any objects representing external resources, e.g., open files or network connections, will become internally inconsistent and fail when their methods are next invoked. After migration these external resources are no longer present. Programmers must therefore code in a migration-aware fashion to prevent this from causing misbehavior in their agents. Migration has global effects. Even though it may only be used in isolated portions of an agent's code, the programmer must be aware of the effect of migration on all of the code. Objects referring to external resources can become unusable, unless the external resources are identical and ubiquitous on all systems or a proxying library is used. Because migration has these side-effects, our migration extension does not allow all previously written Java libraries to be used transparently. Note, however, that migration-aware substitutes for many of the `java.lang` and `java.io` methods could be provided. Much of this remains future work; see Section 7.

# 5 Transformations

We present a series of source code transformations to implement the migration extension of the Java language. These transformations make minimal changes to the original source and provide for migration of programs from one JVM to another.

The initial transformations reduce the full complexity of the Java language to an equally expressive subset, while largely maintaining the readability and the efficiency of the original code. These include moving all variable declarations to the beginning of functions and transforming `for` and `do-while` loops to `while` loops. Other transformations ensure that each function call occurs in a separate statement and that return values are assigned to variables.

The original `main` function is renamed as part of the transformation process, so that a new `main` function can provide run-time support. The original `main` function is transformed just like any other function, so it too can return a migration request. The run-time support code detects this and sends the captured state to the migration target host. The run-time support also initiates state release by restoring the value of the global variables and then passing a non-null `capturedFrame` to the transformed `main` function.

When its added `captureFrame` argument is non-null, i.e., contains a migration frame, the transformed `main` function restores the value of its local variables from that migration frame, in effect consuming that portion of the captured agent state. Then, using a simulated `goto` statement, control flow is transferred to immediately before where the original migration request was returned. That function is then called so that it too can restore its state from the next migration frame. This process continues until the entire call stack has been rebuilt, and all of the variables returned to their pre-migration values. (We may view the transformed `migrate` function as also being called again, except that the presence of its migration frame tells it that it doesn't have to do anything.) At this point the agent has arrived. Its state has been restored, and it resumes normal execution until it makes a new request for migration or exits.

## 5.1 Migration Requests

Each migration request in the original code is transformed into code that builds a migration frame and sets up the parameters to the final communication request to the local server. This is an optimization: abstractly, the `migrate` function is just another function that returns a migration request, and its call site is treated just like any other. See the next section for a description of the general call-site transform.

Any failure in the migration will appear to the original code as a return value from the migration request.

## 5.2 Method Invocations

Every method invocation or function call is transformed to achieve two goals: first, to handle migration requests that are returned by the called function as part of the state capture algorithm, and second, to properly participate in the state release algorithm.

When a function returns a migration request in lieu of its normal return value, it does so by returning its local continuation in a migration frame. At the call site, our transformed code detects the migration request and responds by building and returning the local continuation for the current function. The current function's local continuation includes the local continuation for the called function, the current value of all variables local to the function, and the origin of the migration request. The called function may return normally as well, of course, in which case the flow of control continues as it would have before the transformation. What this means is that all functions logically return a tagged union type, either a migration frame or the normal return value.

To participate in the state release, all call sites are modified to include a (simulated) `goto` label prior to the call itself. This `goto` target is used by the function prolog code inserted by the function declaration transform. During a state release, the input `capturedFrame` parameter is non-null, and this is how the prolog code is triggered. When this happens, the prolog consumes the local continuation for the current function by resetting the local variables to the values they had when the local continuation was captured, and unconditionally branches to the method invocation that had originally returned the migration request. Here, the method is re-invoked with the migration frame for that method. The state release continues recursively until the code arrives at the original `migrate` function, which simply returns success.

The transformation to achieve these two goals is shown in Figure 5. The function declaration transform, which inserts the function prolog needed for the state release algorithm, is show in Figure 6.

**Transform(**"ResultIdentifier = MethodIdentifier ( ArgumentList );"**);**

becomes:

CaptureLabel$N$:
  NewResultIdentifier = MethodIdentifier(capturedFrame, ArgumentList );
  if (NewResultIdentifier.migrateLocation != 0) { `/*` *migration!* `*/`
    captureReturn = new myMigrationResultType($N$,*local-variable-list*);
    return captureReturn;
  } else { `/*` *normal return* `*/`
    ResultIdentifier = NewResultIdentifier.returnedValue;
    NewResultIdentifier = null; capturedFrame = null;
  }

Figure 5: Function call site transformation

## 5.3 Handling Throws and Other Control Flow State

In languages with `try`/`throw` constructs like Java, interprocedural control flow transfers can occur abruptly due to synchronous exceptions, errors, or explicit `throw` statements. Regardless of their source, control flow transfers that bypass our "early" return scheme can cause problems with our state capture technique: if the

```
Transform("ResultType MethodIdentifier(ArgumentList) {
                StatementBlock
          }")
```

becomes:

```
MigrationResultType MethodIdentifier(MigrationResultType capturedFrame, ArgumentList) {
    if (capturedFrame != null) {
        int gotoLoc = capturedFrame.migrationLocation;
        restore state;
        capturedFrame.migrationFrames.setElementAt(null,capturedFrame.frameIndex);
        switch (gotoLoc) {
        case N:
            NextCapturedFrameN = capturedFrame.migrationFrames.elementAt(
                capturedFrame.frameIndex - 1);
            capturedFrame = null;
            goto CaptureLabelN;
        }
    }
    Transform("StatementBlock")
}
```

Figure 6: Function declaration transformation

migration occurs within a `catch` or `finally` block, the flow of control after the migration must mimic that which would have occurred in the absence of migration. State reconstruction is even more difficult within a `finally` block: the code does not have access to the value being thrown (if one is indeed being thrown), which prevents this state from being preserved.

Figure 7 shows how a `try` statement is transformed. In order to handle the problems due to exceptions and other throwables, we transform the `try` statement into equivalent code that explicitly records the current thrown value in a variable whose scope is the entire function. Additionally, we modify the code to catch all throws, not just those explicitly caught by the programmer, so the `Throwable` is recorded in case of a later migration from the `finally` block. Furthermore, to preserve the `finally` semantics, code inserted immediately before `break` and `continue` statements records their target labels so that we can recreate this control flow state at the end of the transformed `finally` block. Here, we re-throw any "uncaught" throws or re-execute `break` or `continue` statements to preserve the original semantics.

# 6 Server Infrastructure

This paper describes our migration transformations. In order for them to be useful, an agent server infrastructure must exist to allow new agents to be created, to accept incoming agents, and to provide resources, both computational and data, for the agents to consume. The exact specification for the agent infrastructure organization is beyond the scope of this paper.

We note that as a matter of good engineering, backward compatibility will be important if agent systems are to become widely used. This is motivated by user expectations of stability: they will expect the agents that they created (or purchased) last year to continue working. Furthermore, simultaneously upgrading all

```
Transform("try {
                T;
           } catch (tType1 t1) {
                C1;
           } catch (tType2 t2) {
                C2; /* etc */
           } finally {
                F;
           }")
becomes:
    try {
        try {
            Transform("T"); /* if T breaks or returns, F still runs */
        } catch (tType1 tmpt1) {
            currThrow = null; t1 = tmpt1; /* t1 now allocated at top of fn */
            Transform("C1"); t1 = null; /* t2 goes out of scope */
                /* may contain migrate or return statements which set and
                return captureReturn; break/continue statements don't, but set
                currBreakOrContinueTarget prior to actually breaking/continuing. */
        } catch (tType2 tmpt2) {
            currThrow = null; t2 = tmpt2;
            Transform("C2"); t2 = null;
        }
    } catch (Throwable t) {
        currThrow = t;
    } finally {
        if (captureReturn == null || captureReturn.migrationLocation == 0) {
            Transform("F"); /* do finally only if not migration return */
            if (captureReturn == null) { /* did not reach here due to transformed return stmt */
                if (currThrow != null) {
                    /* re-throw to any enclosing catch */
                    throw currThrow;
                }
                switch (currBreakOrContinueTarget) {
                case 1: break OuterBreakLabel_1; /* etc */
                }
            }
        }
    }
```

Figure 7: Transforming the `try` statement

agent servers on a widely distributed system borders on the impossible. By incorporating the appropriate version numbers in our agent transfer mechanisms and maintaining an archive of agent library codes, backward compatibility should not be an insurmountable task. Directory services must provide this information, in addition to the more obvious resource availability, cost data, and server security ratings so that agents can decide at run-time whether visiting a server will prove useful or appropriate. The description of the full system awaits our implementation and will be the subject of another report.

# 7 Applicability to Other Languages and Future Work

Our techniques for capturing state apply to other languages than Java and can be used for more than just the migration of autonomous agents. In this section, we describe some other applications and the future direction of our research.

## 7.1 Other Programming Languages

The difficulty of applying our techniques to other languages depends on their language features. There are Java features — notably built-in object serialization — that make extending the language to include the `migrate` operation easier. Other important language characteristics that make our techniques easier to apply are reflection, garbage collection, and strong typing.

For languages with reflection, object serialization can be performed by user-level code that explicitly walks the object graph to generate an external representation of all reachable objects. Such an object list would have all pointers converted to list indices, and primitive types that have default external representations would use those as their serialized form. For objects such as functions, continuations, and closures, creating an external representation is still possible, as we will see below.

For languages without object serialization or reflection that use garbage collection, object serialization can be achieved by extending the garbage collector. In [14], this technique is used to provide persistence to ML. Since persistent data is placed in stable storage, this data is already serialized and compact after log truncation; it is just not directly accessible to the programmer. The approach of garbage collector extension is somewhat less attractive, of course, since distributing changes to compilers or language run-time support code makes its widespread adoption unlikely.

For languages without garbage collection, the task of serializing data is more difficult. In strongly typed languages with pointer comparison, it is possible to walk the graph of all accessible objects and convert it into an external representation. In languages with type inheritance, type tags must be explicitly inserted to provide an *instance-of* operator. This is needed to properly handle the case of a variable of type `T` containing an object that is of some subtype of `T`. When such a value is encountered by the run-time support code's graph traversal, its value must be serialized as its true type.

For weakly typed languages such as C or C++, applying our techniques directly is impossible. This is the same problem faced by conservative garbage collectors for C and C++ [3, 4], except that conservatism forces us send the entire heap. Furthermore, because we cannot distinguish pointers and integers that happen to have the same value, we must ensure that all objects in the heap reside in the same locations before and after migration. In these cases, the pre-compiler techniques presented here are entirely inappropriate, and other process state capture mechanisms should be used [22].

Thread-state migration would only possible if "dangerous" language features such as type casts are avoided. If programmers wrote in a subset of C/C++ where no unsafe type conversions occurred and pointers always point to the start of dynamically allocated memory (rather than the middle, say), it should be possible to provide thread-state migration; using these languages with such artificial restrictions, however, seems pointless.

Perhaps most important for easily applying our techniques is whether external representations exist at all. For languages such as Scheme [6] with first class functions, deriving an external representation of the functional objects is possible. Because such a representation is dependent on the code, we can represent every function with an integer — essentially a function pointer in an abstract name space — as long as the transformed code is used to "interpret" the external representation when the migrated agent is reconstructed from the external representation.

To do this, we would modify the original code to register dynamically created functions (e.g., with `letrec`, `lambda`, or `call-with-current-continuation`) in a dynamic functions table, along with explicit bindings for their free variables (the function's closure) at the time of each function's creation. With this information, the function variables can be recreated from the bindings after migration. When the state capture algorithm traverses the graph of reachable objects, it makes `eq?` comparisons with entries in the dynamic-functions table, and generates table indices as the external representation of functions.

There is a drawback to this approach. Because we maintain a reference to the dynamically generated function in the dynamic-functions table, garbage collection of these functions is prevented. Further automatic modifications to the code are needed to maintain reference counts, so that the dynamic-functions table can eliminate otherwise unreachable functions. (If Scheme had an equivalent of "weak references" of Java, this would be much simpler.)

Handling first class continuations introduces a little more complexity and a lot of overhead. To allow explicit access to variables in the stack frames (`let`, `letrec`, etc.), functions are modified in the same way as we did for Java. We cannot introduce an extra argument when calling continuations, however, so we simply wrap the actual argument along with our migration frame argument in a tuple. The creation of the continuation itself in `call-with-current-continuation` becomes much more expensive. The continuation must be in an explicit representation that can be captured, since the function called by `call-with-current--continuation` may invoke `migrate`. To obtain an explicit, user-level representation for the continuation, the function invoking the `call-with-current-continuation` must explicitly capture the list of migration frames from the current call stack via special "returns". This implies that every function call is now a `call-with-current-continuation` to permit non-local returns. After capturing the explicit representation of the continuation, we restore the program state via the state release mechanism to get control back to the function containing the original `call-with-current-continuation`. With the explicit call stack (in the form of a migration frames list) in hand, the original `call-with-current-continuation` may be invoked with that as an extra argument.

There are many language specific details that need to be worked out for each language to which our migration techniques are to be applied. Exactly how explicit continuations and closures are captured varies from language to language, and details of how to implement our techniques vary according to interactions with language features. Nonetheless, our techniques are sufficiently general that they may be applied to many languages other than Java.

## 7.2 Other Applications

The generality of our techniques implies that it can be used for other applications than mobile agents.

One obvious application is to add continuations and closures to strongly typed languages that do not already have these features. While our techniques are somewhat expensive, using them might be tolerable in programs where the use of the continuation passing style is dynamically infrequent.

Another application is process checkpointing. Here, the process is migrating via a very slow communication channel — the disk — where the originating and destination hosts are one and the same. Such checkpointing is typesafe insofar as object serialization is typesafe, and thus provides a form of typesafe fault tolerance.

A variation on the checkpointing theme is the use of these techniques to allow systems to support a large number of stored procedures. While trigger conditions for the stored procedures must be in main memory and so be efficiently represented, the stored procedures themselves require little special treatment. Because the stored procedures need not consume any resources other than secondary storage while not active (aside from the representation of their trigger conditions), systems can potentially support more stored procedures than otherwise. A drawback is that the state release and deserialization can be expensive, and simpler ways of checkpointing — which are typically not typesafe — might be more appropriate.

### 7.3 Future Work

In this paper, we described the design of syntactic transforms mainly targeted to the migration of agents. Immediate needs for a complete migration system in addition to the basic language level transformations are migration-aware libraries and the design of language extensions (if any) needed to support such libraries.

In order to provide an easy environment for agent programmers to work in, library authors need to have their libraries notice that a migration is about to happen or has happened, and do whatever migration-related bookkeeping is required, e.g., setting up proxies for external resources that did not migrate with the program. To this end, we have designed two complementary mechanisms for the library author: the ability to mark objects as "sessile" or "motile", and methods for registering callbacks so that library code can perform the necessary bookkeeping. The details of this design are currently being worked on and will be the subject of a future paper.

Progress is being made on the pre-compiler implementing our techniques. Experiments with it to determine the exact cost of various alternative approaches — e.g., always using heap-allocated activation frames, the interaction with server-side run-time support code, etc. — should provide interesting data, and we may wish to revisit some of our design choices as we gain experience with the use of our system.

## 8 Conclusion

We presented a method for extending programming languages — in particular, to add a `migrate` function — by a series of syntactic transformations. These transformations give an explicit user-level representation of procedure activation frames, providing local and full continuations as extended language features. Moreover, by applying these transforms to Java, we showed that mobile agent migration can be implemented by extracting continuations as an explicit data structure that can then be sent from one machine to another via object serialization. Furthermore, our syntactic transformations are general: the same basic ideas can be applied to other programming languages and implemented using either pre-compilers or binary-to-binary (or bytecode) translators.

We believe that our migration extension to the Java language, achieved by applying our syntactic transformations via a pre-compiler, provides the appropriate level of abstraction — thread-state migration — for mobile agent migration. Furthermore, our technique reduces the complexity of agent server code by isolating the complexity of migration to user code, reducing the correctness assumptions that must be made about servers. This has both operational and security implications, since users can upgrade their agents as needed, independently of the myriad of servers on the Internet; this is much easier to do than asking every agent server operator to perform upgrades.

Regardless of whether our approach is implemented via a pre-compiler or a bytecode postprocessor, the resultant bytecodes are compatible with existing JVMs. When new, more efficient JVMs or just-in-time compilers become available, we can immediately take advantage of them. When the pre-compiler approach is used, we can also immediately take advantage of new optimizing compilers as they are released. Thus, our transformational approach is largely orthogonal to optimizations, with the minor caveat that the transformations might make the compiler less effective in applying optimizations. Optimizing compilers

```
class C {
        retType thrower(int[] array) throws tooStingyForMeException {
                retType result;
                if (array[5] == 5) {
                        throw new tooStingyForMeException(targetHost);
                } /* result gets initialized in elided code */
                return result;
        }
        retType caller(int[] in_array) {
                retType result; int k = 0;
                try {
                        result = thrower(in_array);
                        /* Elided code */
                } catch (tooStingyForMeException bad_val) {
                        k = migrate(bad_val.targetHost);
                } finally {
                        System.out.println("Value: " + in_array[5] + "k: " + k);
                }
                return result;
        }
};
```

Figure 8: Transforming the `try` statement: the original code

can make control flow recovery and analysis for bytecode translators more difficult, and the result of the interaction of these components on performance is less clear.

Finally, our code transformations may be included as a new component in an existing Java-based mobile agent system. While the semantics of our migration techniques makes clean, easy-to-use, migration-aware libraries a new design goal, existing agent APIs can still be used as long as all library objects are treated as sessile objects — writing agents using our migration transform technology and existing agent infrastructures should be easier. More data on the true impact of our approach awaits the completion of the pre-compiler and subsequent experimentation.

# A Detailed Example: Throwables

This appendix contains a detailed example of the transformations required to allow migration while properly handling Java's `Throwables`. As before, our code is written in a special dialect of Java where variable declarations occur before executable statements, and forward `goto` statements are allowed. The example is presented in sequence, showing what the code looks like in intermediate steps. First, Figure 8 shows the original code. Note that the code is not designed to actually accomplish anything meaningful, but to aid in our exposition. Next, Figure 9 shows the code after we've made "`Throwable` hand-off" explicit: all Java `Throwables` are caught and re-thrown in the `finally` block. This code with the integrated migration and `try` transformations is in Figure 11, with the supporting return types shown in Figure 10.

We omit the initial transformation that moves all variable declarations before executable statements, and the final transformation that adds flags and converts the `goto` statements into `goto`-free "structured" code.

```
class C {
	retType thrower(int[] array) throws tooStingyForMeException {
		retType result;
		if (array[5] == 5) {
			throw new tooStingyForMeException(targetHost);
		} /* result gets initialized in elided code */
		return result;
	}
	retType caller(int[] in_array) throws Throwable {
		Throwable currThrow = null;
		tooStingyForMeException bad_val;
		retType result; int k = 0;
		try {
			try {
				result = thrower(in_array);
				/* Elided code */
			} catch (tooStingyForMeException t) {
				currThrow = null; bad_val = t;
				k = migrate(bad_val.targetHost);
				t = null;
			}
		} catch (Throwable t) {
			currThrow = t;
		} finally {
			System.out.println("Value: " + in_array[5] + "k: " + k);
			if (currThrow != null) {
				throw currThrow;
			}
			/* no break or continue */
		}
		return result;
	}
};
```

Figure 9: Transforming the `try` statement: resultant code

```
class capture_throwable_retType2 implements Serializable { /* thrower */
	int migrateLocation; int migrateStatus; Vector migrationFrames; /* migration return */
	int frameIndex; hostInfo targetHost;
	Throwable currThrow; /* unhandled throwable */
	retType returnedValue; /* normal return */
	int[] array; /* arglist */
	retType result; /* local */
};

class capture_throwable_retType implements Serializable { /* caller */
	int migrateLocation; int migrateStatus; Vector migrationFrames; /* migration return */
	int frameIndex; hostInfo targetHost;
	Throwable currThrow; /* unhandled throwable */
	retType returnedValue; /* normal return */
	int[] in_array; /* arglist */
	int k; tooStingyForMeException bad_val; /* locals */
	retType result; /* local */
};
```

Figure 10: New Classes created for migration

```
class C {
        capture_throwable_retType caller(capture_throwable_retType capturedFrame, int[] in_array) {
                retType result; int k;
                tooStingyForMeException bad_val; /* Brought from catch clause */
                capture_throwable_retType captureReturn; int gotoLoc;
                Throwable currThrow = null; /* Throwable currently in flight */
                capture_throwable_retType2 migrMethFrame = null; /* new return type of thrower */
                if (capturedFrame != null) {
                        gotoLoc = capturedFrame.migrationLocation;
                        in_array = capturedFrame.in_array;
                        currThrow = capturedFrame.currThrow;
                        k = capturedFrame.k; bad_val = capturedFrame.bad_val;
                        result = capturedFrame.result;
                        capturedFrame.migrationFrames.setElementAt(null,capturedFrame.frameIndex);
                        switch (gotoLoc) {
                        case 1: migrMethFrame = (capture_throwable_retType2)
                                                capturedFrame.elementAt(capturedFrame.frameIndex - 1);
                                capturedFrame = null; /* allow gc */
                                goto CaptureLabel1;
                        case 2: k = capturedFrame.migrateStatus; /* k = migrate(bad_val.targetHost) */
                                capturedFrame = null; /* allow gc */
                                goto CaptureLabel2;
                        }
                }
                try { /* Around the try-catch blocks */
                        try { /* Around the try block */
CaptureLabel1:                  ;
                                migrMethFrame = thrower(migrMethFrame, in_array);
                                if (migrMethFrame.migrationLocation != 0) { /* a migrate request */
                                        captureReturn = new capture_throwable_retType();
                                        captureReturn.migrateLocation = 1;
                                        captureReturn.in_array = in_array;
                                        captureReturn.result = result; /* local */
                                        captureReturn.k = k;
                                        captureReturn.currThrow = null;
                                        captureReturn.bad_val = null; /* not in scope */
                                        captureReturn.migrationFrames = migrMethFrame.migrationFrames;
                                        captureReturn.frameIndex = migrMethFrame.frameIndex + 1;
                                        captureReturn.migrationFrames.setElementAt(captureReturn,
                                                                        captureReturn.frameIndex);
                                        return captureReturn;
                                } else { /* a normal return */
                                        result = migrMethFrame.returnedValue; migrMethFrame = null;
                                }
                                /* Elided code */
                        /* Continues on next page */
```

Figure 11: Fully Integrated Example: `try` and `migrate` transforms (caller only)

```
			} catch (tooStingyForMeException tmp1) {
				currThrow = null; bad_val = tmp1;
				/* migrate(bad_val.targetHost); */
				captureReturn = new capture_throwable_retType();
				captureReturn.migrateLocation = 2;
				captureReturn.targethost = (bad_val.targetHost);
				captureReturn.in_array = in_array;
				captureReturn.result = result;
				captureReturn.k = k;
				captureReturn.returnedValue = null;
				captureReturn.currThrow = currThrow;
				captureReturn.bad_val = bad_val;
				captureReturn.migrationFrames = new Vector();
				captureReturn.frameIndex = 0;
				captureReturn.migrationFrames.setElementAt(captureReturn,0);
				return captureReturn;
CaptureLabel2:			;
				bad_val = null; /* bad_val goes out of scope */
			}
		} catch (Throwable generic_throwable) {
			currThrow = generic_throwable;
		} finally {
			if (captureReturn == null || captureReturn.migrationLocation == 0) {
				System.out.println("Value: " + in_array[5] + "k: " + k);
				if (captureReturn == null) {
					if (currThrow != null) {
						/* re-throw to any enclosing catch */
						throw currThrow;
					}
					/* no breaks or continues */
				}
			}
		}
		captureReturn = new capture_throwable_retType();
		captureReturn.returnedValue = result;
		captureReturn.migrateLocation = 0;
		return captureReturn;
	}
};
```

Figure 11 (cont): Fully Integrated Example: `try` and `migrate` transforms (caller only)

## References


[1] Stephane Belmon and Bennet Yee. Mobile agents and intellectual property protection. In *Proceedings of the Second International Workshop on Mobile Agents 98*, Lecture Notes in Computer Science. Springer-Verlag, September 1998. To appear.

[2] Nick Benton, Andrew Kennedy, and George Russell. *The MLJ User Guide*. Persimmon IT, Inc, Cambridge, UK, May 1998.

[3] Hans Boehm. Space efficient conservative garbage collection. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, number 28, 6 in SIGPLAN Notices, pages 197–206, June 1993.

[4] Hans Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, pages 807–820, September 1988.

[5] Per Bothner. Kawa — compiling dynamic languages to the java vm. In *Proceedings of the USENIX 1998 Technical Conference, FREENIX Track*, New Orleans, LA, June 1998. USENIX.

[6] William Clinger and Jonathan Rees, eds. Revised$^4$ report on the algorithmic language scheme. *ACM Lisp Pointers*, 4(3), July 1991.

[7] General Magic, Inc. *Introduction to the Odyssey API*. 20 North Mary Avenue, Sunnyvale, CA 94086. Manual published as `http://www.generalmagic.com/agents/odysseyIntro.pdf`.

[8] General Magic, Inc. An introduction to safety and security in Telescript, 1995. White paper published as `http://www.generalmagic.com/Telescript/security.html`.

[9] Jonathan C. Hardwick and Jay Sipelstein. Java as an intermediate language. Technical Report CMU-CS-96-161, Carnegie Mellon University, Pittsburgh, PA, August 1996.

[10] Colin G. Harrison, David M. Chess, and Aaron Kershenbaum. Mobile agents: Are they a good idea? *Lecture Notes in Computer Science*, 1222:25–47, 1997.

[11] Dag Johansen, Fred B. Schneider, and Robbert van Renesse. What TACOMA taught us. In Dejan Milojicic, Frederick Douglis, and Richard Wheeler, editors, *Mobility, Mobile Agents and Process Migration — An edited Collection*. Addison Wesley, 1998. To appear.

[12] Danny B. Lange and Daniel T. Chang. IBM Aglets workbench. IBM Tokyo Research White paper at `http://www.trl.ibm.co.jp/aglets/whitepaper.htm`, September 1996.

[13] Jacob Y. Levy and John K. Ousterhout. A safe Tcl toolkit for electronic meeting places (extended abstract). In *Proceedings of the First USENIX Workshop in Electronic Commerce*, pages 133–135, July 1995.

[14] James O'Toole, Scott Nettles, and David Gifford. Concurrent compacting garbage collection of a persistent heap. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 161–174, December 1993.

[15] Holger Peine and Torsten Stolpmann. The architecture of the ARA platform for mobile agents. In Kurt Rothermel and Radu Popescu-Zeletin, editors, *Proceedings of the First International Workshop on Mobile Agents*, volume 1219 of *Lecture Notes in Computer Science*. Springer-Verlag, April 1997. ISBN 3-540-62803-7.

[16] Fred B. Schneider. Towards fault-tolerant and secure agentry. In *Proceedings of the 11th International Workshop on Distributed Algorithms*, Saarbucken, Germany, September 1997.

[17] Markus Straßer, Joachim Baumann, and Fritz Hohl. Mole — a Java based agent system. In *Proceedings of the ECOOP '96 Workshop on Mobile Object System*, 1996.

[18] David L. Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, and Gary J. Minden. A survey of active network research. *IEEE Communications Magazine*, pages 80–86, Jan 1997.

[19] Chelliah Thirunavukkarasu, Tim Finin, and James Mayfield. Secret agents — a security architecture for the kqml agent communication language. In *Proceedings of the CIKM '95 Intelligent Information Agents Workshop*, Baltimore, MA, December 1995.

[20] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel. The LOCUS distributed operating system. In *Proceedings of the Ninth Symposium on Operating System Principles*, pages 49–70. ACM, 1983.

[21] J. E. White. Mobile agents. In J. Bradshaw, editor, *Software Agents*. AAAI Press and MIT Press, 1996.

[22] Bennet S. Yee. The portable `save_world` process checkpointing package, 1986. Software package available from `http://www.cs.ucsd.edu/~bsy/pub/save_world.tar`.

[23] Bennet S. Yee. A sanctuary for mobile agents. In *Proceedings of the DARPA Workshop on Foundations for Secure Mobile Code*, March 1997.