# Dynamic Code Updates

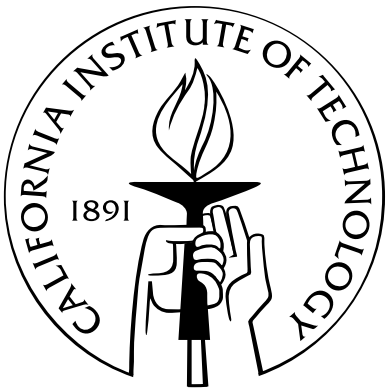
Thesis by

Michael Maire

In Partial Fulfillment of the Requirements

for the Degree of

B.S. in Electrical and Computer Engineering




California Institute of Technology

Pasadena, California

2003

(Submitted June 4, 2003)

# Acknowledgements

I would like to thank my advisor, Professor Jason Hickey, for his guidance and insight. I would also like to thank my second reader, Professor K. Mani Chandy, for his help in revising this thesis.

# Abstract

We present a system for dynamically updating the structure of a running program in a type-safe manner. This capability allows bug fixes and software upgrades to be applied to critical services with minimal interruption. Unlike previous dynamic update systems, arbitrary changes to the program's source code are permitted and an update can be successfully applied at any point during execution. At the same time, we supply the programmer with a powerful model for enforcing the semantic consistency of updates.

# Contents

# List of Figures

# Chapter 1

# Introduction

## 1.1 Motivation

The main challenge in constructing a system for dynamically altering a program is designing one that is flexible while also guaranteeing that program modifications preserve type safety and desired semantic properties. We present an update methodology for object-oriented programs that offers significant advantages in these areas over previous work. At the same time, we provide a simple, powerful programming model for writing updateable code.

In order to demonstrate the feasibility of this methodology, we extended the Caltech Mojave compiler project [7] to support updateable Java programs. Grafting the static analysis and dynamic runtime facilities of the update system onto a complete Java compiler produced a system that can be used in creating practical updateable software.

## 1.2 Applications

A number of software applications must function continuously in order to provide critical services. Banks, corporate web sites, and air traffic control systems all require software solutions with high availability. Software downtime has the potential to cost money or even lives. However, any critical software system will eventually require maintenance, in the form of bug fixes and possibly feature enhancements. The ability to perform this maintenance with minimal service interruption is extremely important.

Dynamic update systems provide a way of maintaining and enhancing running programs without loss of state information. Rather than terminating an executing program and starting a newer version of the program, our dynamic update system momentarily pauses the program, coerces its state into

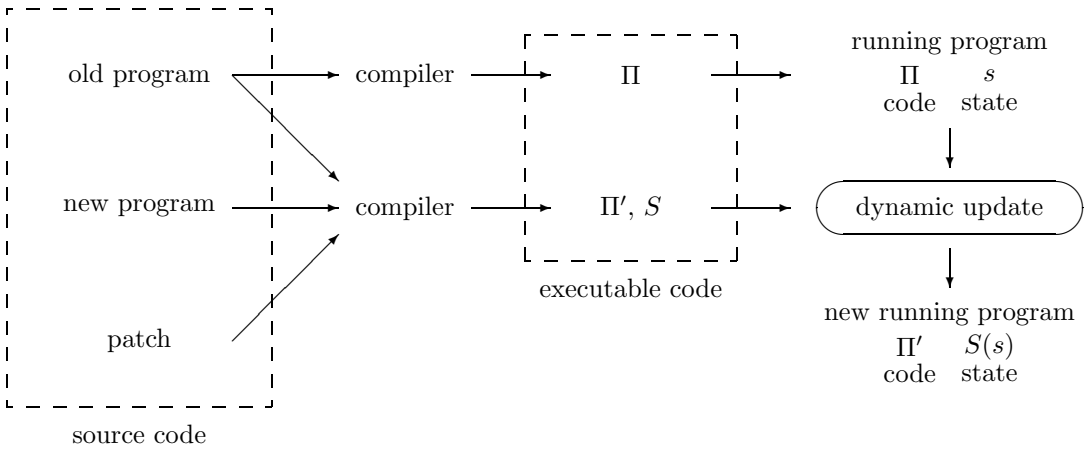



Figure 1.1: Overview of a generic dynamic update system.

one compatible with the new version, and resumes execution with the new program. The amount of coercion necessary roughly reflects the size of the difference between the two program versions. The coercion of the program state allows upgrades to be performed without, for example, disconnecting clients or discarding unsaved data. This can translate into a significant reduction in downtime when maintaining critical systems.

## 1.3 Previous Work

Significant effort has been placed into developing systems for dynamically updating computer programs [3, 4, 5, 8, 9, 10, 11, 12, 13, 14]. Gupta [4] gives an overview of many of these systems as well as presenting a mathematical framework for describing online program modification. It is useful to review part of this theoretical background before discussion individual update systems.

A running program, or process, consists of two components, the code being executed, $\Pi$, and the current state, $s$. The state includes all of the process's live data as well as a position defining the next command to be executed. Dynamically updating a process running program $\Pi$ to one running a newer program version, $\Pi'$, necessitates replacing code $\Pi$ with $\Pi'$ and transitioning state $s$ of $\Pi$ to a state $s'$ of $\Pi'$ [4]. A dynamic update system must provide means of specifying a state transformer, $S$, which maps states $s$ of $\Pi$ to states $s'$ of $\Pi'$ in a manner that produces desirable behavior. Effectively designing this state transformer is the primary challenge in building an update system. Figure 1.1 outlines the update procedure for a generic state transformer.

Practical update systems must place restrictions on the allowable state transformations, however

the appropriate set of restrictions is not clear. Gupta suggests that state mapping $S$ is valid if, for all states $s$ of $\Pi$, program $\Pi'$ starting from state $s' = S(s)$ enters a reachable state $s''$ of $\Pi'$ after a finite number of steps [4]. A state $s''$ of $\Pi'$ is reachable if and only if there is some input which causes $\Pi'$ to enter state $s''$ in a finite number of steps, starting from its initial state. Intuitively, this restriction requires that the system eventually reach a state that is as if the new program had been running all along, albeit on possibly different inputs.

Unfortunately, this characterization of valid states is overly restrictive. The states reachable by running $\Pi'$ might not correspond to the meaningful states on which $\Pi'$ can be executing. Some such reachable states may be meaningless, perhaps because the input used to reach them would never occur. More importantly, it may be meaningful and useful to run $\Pi'$ on states not reachable from its initial state. The programmer, rather than the update methodology, should make this semantic judgment.

Furthermore, using the definition given above, the question of whether $\Pi'$ enters a reachable state when started from $s'$ is in general undecidable. A possible solution to this decidability problem is to restrict the set of allowed changes to the program. It is sufficient to require $\Pi'$ to be a *functional enhancement* of $\Pi$ with respect to $S$, such that there is a trivial correspondence between the execution path of $\Pi$ taken to reach $s$ and the execution path of $\Pi'$ that can be taken to reach $S(s)$ [4]. However, this significantly restricts the kinds of changes one can make to the program, potentially ruling out simple bug fixes.

While the above proposals are too restrictive, any effort to weaken them should at least guarantee that the state transformer produces new states that are valid from the programmer's point of view.

In order to simplify the process of specifying the state transformation, it often makes sense to limit its domain. By guaranteeing certain properties about the state $s$ in which the old program is halted, it may be easier to specify a state transformer $S$ that always produces a valid new state $S(s)$. This approach is taken by virtually all dynamic update systems in existence.

The simplest solution for restricting the program state at the time of change is to designate specific reconfiguration points within the program. Hicks [8, 9] makes this choice, allowing an update to be performed only when execution reaches a reconfiguration point. An updateable program must be designed from the start to request its own updates.

A more flexible, but more complex, solution is to define conditions the state must satisfy before an update is applied. This is typically done by specifying a set of functions or methods that cannot be executing at the time of the change. This strategy attempts to make two guarantees. First, all

data that could be accessed during the state transformation is in a consistent state. For example, the program is not in the middle of rearranging the pointers of a linked list that the state transformer will need to use. Second, execution can be resumed exactly where it was stopped. There is a trivial way to map the instruction pointer from the old program to the new program.

As Malabarba et. al. [12] point out, this set of functions or methods should minimally be those that have been altered in the new program version. This is necessary for ensuring the second condition above. There is no automatic way to determine what other methods should not be running at the time of update as this is a semantic question. Sunil [13], for example, expects the programmer to manually specify these requirements. Unfortunately, when an update is requested, it is entirely possible for the program to be executing functions in this forbidden set.

Frieder and Segal [3] solve this issue by allowing the program to continue running, while replacing the old version of a function's code with the new version at the earliest possible time. The system updates a function as soon as the program returns from all calls to that function. As a result, at any time, the program may consist of a mix of the old and new versions of the code. In addition, there is no guarantee the update process will ever complete. For example, an old function that should be updated might never return. Applying multiple updates in this manner could result in a mix of program versions about which reasoning is difficult.

An analogous approach for object-oriented systems is to define conditions for each class under which it is acceptable to update instances of that class. For example, an object instance's data is restructured and its methods are upgraded to new versions once none of the class's old methods are executing on that instance. This scenario is similar to that used by Tang [14] to evolve the implementation of components while leaving their interface unchanged. Hjalmtysson and Gray [10] also permit multiple versions of the same class to coexist, although they rely on the program to request updating of older instances, rather than automatically transitioning when possible. These schemes still suffer the drawback that code from old program versions may remain active even after many updates.

A central theme of the work discussed above is the effort to define an appropriate state transition process for dynamic update systems. Proposed solutions simplified this process by restricting the changes that could be made to the original program, by restricting the states from which a process could be updated, or by abandoning the guarantee on complete application of the update.

## 1.4 Outline

The remainder of this thesis is structured as follows. Chapter 2 presents the main challenges considered in designing our dynamic update methodology. This includes important requirements for any update system, such as type safety and ease of use, as well as areas in which improvements over previous systems are desired.

Chapter 3 draws the framework for an update system which answers these challenges. We present a methodology significantly more powerful and more flexible than its predecessors. Our framework makes two guarantees that set it apart from other dynamic update systems. First, an update can be successfully applied on demand, without waiting for the program to enter a special state. Second, any program, even one designed without update capability in mind, can be updated to any other program, even one that is completely different.

These guarantees come at a minimum cost in the form of runtime overhead. In addition, there is a chance that some amount of computation will be lost and restarted as a result of an update. This penalty is roughly proportional to the degree of change made to the program source code and is usually insignificant. Chapter 3 discusses this runtime system and also describes the tools available to the programmer for controlling and customizing the update process.

Chapter 4 details the implementation of the update system on top of an existing Java compiler. We outline the changes required in the intermediate program representations, the backend, and the runtime.

Finally, Chapter 5 gives a summary of the work and discusses future research possibilities, such as increased programmer control over the update mechanism and support for multithreaded programs.

# Chapter 2

# Design Challenges

## 2.1 Type Safety

A type system is a form of annotation used to specify properties of a program. For example, a programmer may annotate a particular variable with the "int" keyword to indicate that variable represents an integer value, rather than a pointer to an object. Type systems are useful because they help prevent common programming errors, such as attempting to use an integer as a pointer. A type-safe language is one in which the type system is used to make two important guarantees about program behavior. First, the types of program elements do not change as the program executes. Second, a value of one type will never be used where a value of an incompatible type is required. This property ensures that a program can continue to be evaluated until it has been reduced to a single value. The program will not get stuck or cause a segmentation fault.

Type safety is an integral component of modern programming languages such as Java. An unsafe update system built for a type-safe language would negate the language's key benefits. The assurances on program behavior provided by type safety are likely to be especially important in applications where dynamic update capability is desired. The design of long running critical systems usually places greater value on stability and correctness than on any marginal speed improvements which might be gained from sacrificing type safety.

Using the abstract model of state transition introduced in Chapter 1, the following conditions are sufficient to guarantee the type safety of a dynamic update:

1. The old program, $\Pi$, is type-safe.
2. The new program, $\Pi'$, is type-safe.
3. The state transformer, $S$, is type-safe.

4. Transforming a state $s$ of $\Pi$ results in a state $s' = S(s)$ of $\Pi'$ such that running $\Pi'$ from state $s'$ is type-safe.

The old program is either an original program written in the type-safe source language, or a dynamically updated version of some original program. In the first case, the compiler enforces type safety. In the second case, the old program is type-safe provided the update system respects type safety. Hence, by induction, condition (1) above is satisfied provided conditions (2)-(4) are satisfied.

Condition (2) can be simplified by requiring $\Pi'$ to be the result of compiling a program written in the type-safe source language. In Figure 1.1, $\Pi'$ would depend only on the source code of the new program and not on the patch or old program source code. Thus, (2) is satisfied as the compiler enforces type safety.

By making this design choice, we rule out mixing code from multiple versions as allowed by other update systems [10, 14]. After an update, only code from the most recent version will be running. This simplifies future updates as the programmer need not be concerned about compatibility with ancient versions of the program. However, this also rules out the best-effort update strategy used by these other systems. To justify this trade off, we will later provide a reasonable update strategy which guarantees complete transition into the new program version.

Conditions (3) and (4) are linked to this update strategy. The state transformer depends both on the patch code and the runtime utilities for applying the patch. Like most update systems for object-oriented languages [10, 12, 13, 14], we take classes to be the basic unit of change. The patch code defines, for each changed class, a function to coerce an instance of the old class version to an instance of the new class version. The compiler can type check this code, guaranteeing the coercions are type-safe and satisfying condition (3).

However, type-safe patch code is not sufficient to guarantee condition (4). The patch code defines the entire state transformation with the exception of the point at which the old program is stopped and the point at which the new program is resumed. For example, stopping the old program at a call to a method that expects an integer argument and resuming the new program, with this argument, at a method that expects an object, is clearly unsafe.

Our system guarantees (4) by ensuring compatibility between the transition points in the two program versions. In order to accomplish this, we convert all programs to continuation passing style (CPS) as part of the compilation process [2]. CPS wraps the code inside branches and loops into separate functions. Every function receives an extra argument, the continuation, indicating the function to be called after its own completion. Thus, each branch point in the program is reduced

to a function call, referred to as a tailcall. Moreover, the entire program state is captured by the arguments to a tailcall because the tailcall does not return but rather calls one of its arguments upon completion.

Tailcalls thus provide a natural transition point at which an update can be performed. By halting the old program at a tailcall, all objects that need to be updated are accessible through its arguments. Furthermore, we can guarantee type safety by resuming the new program on a tailcall which takes the same arguments as the old tailcall, with the difference that their types are the updated types of the old arguments.

Note that choosing a correct state in which to perform an update is also a semantic concern. In our system, the compiler finds the type-safe correspondences between tailcalls in the two program versions. However, we also allow the programmer to specify additional restrictions on the states in which an update can be applied. The timing and semantics sections which follow outline the rational for such restrictions.

## 2.2 Timing

As discussed in Chapter 1, placing restrictions on when an update is permissible can simplify the process of writing correct patch code. In our object-oriented update model, we can use such restrictions to guarantee that each object instance to be updated is in a consistent state. For example, when updating a List class, a desirable guarantee is that the program is not in the middle of adding or deleting elements from any List instance. Without this assurance, the patch code for updating a List instance could be enormously complicated. Moreover, each List instance would have to store additional information, allowing access to a consistent view of the list regardless of the program state.

Reconfiguration points [8, 9] provide only a cumbersome method of specifying these restrictions. While specifying global points at which a program requests to be updated may work for a simple web server, it is unlikely to scale well for larger or more complex applications. For such programs, designing reconfiguration points to account for all future update needs is more difficult. In addition, guaranteeing that a complex program periodically encounters a reconfiguration point may be difficult. The delay between between such encounters may aggravate efforts to apply an urgent patch.

The above difficulties arise partly because reconfiguration points are overly restrictive. They force updates to occur only when it is acceptable to change any class the programmer could conceivably

want to update. However, any particular update is likely to change only a small portion of the program. Thus, the set of acceptable points at which to update is usually much larger than the set of specified reconfiguration points.

Specifying timing restrictions as part of the update process, in the form of methods that cannot be on the stack, is less restrictive than using reconfiguration points [13]. However, this strategy is still too restrictive. For example, an entire method might be excluded when in fact only a small block of the code within the method violates a timing requirement. Even if only one of possibly many execution paths within the method violates a requirement, the entire method is restricted. Furthermore, the task of manually specifying restricted methods for a particular update is more involved and hence more error prone than specifying reconfiguration points.

The goal in specifying timing restrictions is to permit the update system to determine, at any point during the program's execution, which object instances are in an inconsistent state. Our update system provides an exact answer to this question by modifying the runtime to mark instances in inconsistent states. We provide the programmer with the ability to control this procedure in the source code. Our system can also integrate additional consistency requirements specified at update time but left out of the original version of the code.

## 2.3 Semantics

Coupled to the problem of determining the time at which an update should occur is the task of ensuring the update produces a meaningful state transition. Timing requirements guarantee the consistency of the state $s$ in which the old program is halted. However, we still require a guarantee on the consistency of the transformed state $S(s)$ on which the new program is resumed.

Gupta showed the ideal guarantee, that resumption on state $S(s)$ leads the new program to behave as if it had been running all along, is uncomputable [4]. Acar et. al. demonstrate that for purely functional programs, a general method exists for propagating a change in the program's input to the corresponding change in the output [1]. However, an update is usually not expressible as only a change to an algorithm's input. In general, updates alter the algorithm itself. Furthermore, the programs for which dynamic updates are valuable are unlikely to be purely functional algorithms.

By ruling out the use of reconfiguration points, we rule out the simplest model for enforcing meaningful updates. We choose not to place the burden on the programmer to design the update mechanism from the start. As outlined in the type safety section above, automatically identifying tailcalls with the same argument types between the two program versions gives the set of possible

points at which to transition between program versions. By adopting the convention that methods with the same name and type have the same purpose in each program, we can automatically restrict this set to the set of semantically correct update points. This convention does not restrict the programmer, as methods can be renamed in the new program version if they do not uphold this semantic correspondence.

Using this definition of semantic consistency, the update process must stop and resume programs at a method or function call defined by the programmer in both versions of the source code. By transforming programs to continuation passing form as part of the compilation process, it is guaranteed that a running program will eventually reach a tailcall. Loop bodies are wrapped into functions, to which a tailcall is made every loop iteration. By checking for an update request at the beginning of each tailcall, we guarantee that even programs inside an infinite loop will respond to the update system. Unfortunately, there is no guarantee the tailcall at which a program is stopped will correspond a call to a programmer defined method in the source code.

Even if an update request triggers a program to stop at a method call, the new program version might not contain a corresponding method call. Since our update methodology supports arbitrary changes, one can update a program with an entirely different program. In this case, the only semantically consistent update point found by the compiler would be the call to the main function starting each program. Once the old program had begun execution, it could never be stopped in a state deemed valid for update application.

In order to solve the update point correspondence problem, we provide a mechanism to roll back the state of a stopped program to the last semantically acceptable method or function call. With this mechanism, we simultaneously solve the problem of enforcing the consistency of object instances at update time. The system rolls back the program state to one for which no objects scheduled for update by the state transformer are marked as inconsistent.

## 2.4 Ease of Use

The final design challenge is to package the complex solutions to the problems of type safety, timing, and semantics into a simple, intuitive system for the programmer. Crafting updateable programs should not be significantly harder than normal programming tasks. Understanding the application of updates and the behavior of updated programs should be a straightforward process.

We simplify the task of writing updateable programs in several ways. Our methodology separates development of patch code from development of the new program version. This permits the new

version to be executed in a standalone manner, independent of previous versions. The patch is only needed to update older systems that have already been deployed. We also provide an intuitive model for specifying consistency requirements which possesses advantages over reconfiguration points in terms of both flexibility and scalability.

Through enhancements to the compiler and runtime system, we automate the task of ensuring the programmer-specified consistency requirements are satisfied. Rollback capability allows us to guarantee successful application of an update, without the drawbacks of manually specifying reconfiguration points.

# Chapter 3

# Framework

## 3.1 Object Based Update Model

Our update system is object-oriented in the sense that object instances are the basic units upon which a dynamic update acts. We restrict the state transformation $S$ to be composed of a series of transformations, $\{S_1, S_2, \ldots, S_n\}$, where $S_i$ is the transformation for objects of class $i$. Specifically, $S_i$ is a function which coerces an instance of the old version of class $i$ into an instance of the new version of class $i$. Applying the state transformation $S$ to a state $s$ of a program is equivalent to applying transformation $S_i$ to each instance of class $i$ contained in $s$.

We make no guarantee concerning the order in which instances in state $s$ are updated. A valid state transformation should produce the same result regardless of the order in which the $S_i$'s are applied to object instances. In addition, we guarantee the update process completes successfully on the condition that each $S_i$ terminates. The programmer is responsible for writing patch code within these limitations. However, neither of these restrictions is severe as the state transformation typically serves only to update an object's internal representation.

No restrictions are placed on the kind of changes a programmer may make to a class definition. Our system supports addition and removal of both fields and methods. The programmer may also remove entire classes or add new classes. The compiler automatically determines the correspondence between the classes in two different programs based on class name. Given two versions of a particular class, the compiler determines correspondences for fields and methods based on both name and type. In order to match, these program elements must have exactly the same names and types.

## 3.2 Semantic Preservation

### 3.2.1 Synchronization

To assist the programmer in writing a correct patch, we guarantee an instance of class $i$ will be in a consistent state when passed as an argument to $S_i$. As previously discussed, the definition of "consistent" is left to the programmer as it is a semantic property. We support arbitrary definitions of consistency by requiring the program to inform the runtime whenever an instance's consistency status changes.

The programmer conveys this information to the runtime by encapsulating any code within which an object enters an inconsistent state inside a synchronization block on that object. For example, if $\alpha$, an instance of class $A$, could be placed in an inconsistent state by code segment $\pi$, then the source code should declare this code segment as: synchronized($\alpha$) { $\pi$ }.

We borrow the syntax of synchronization locks for multithreaded programs precisely because locking an object from the update system in a single threaded program is analogous to locking an object from other threads in a multithreaded program. Synchronization can be used to prevent two threads from simultaneously attempting to rearrange the internal structure of an object. If one views the dynamic update system as a special program thread, executing in parallel with the original program, then this form of access control is exactly that required to enforce consistency. The dynamic update thread must be prevented from updating object internals in use by the main program thread.

### 3.2.2 State Rollback

As outlined in Chapter 2, when an update is requested, a program is rolled back to the most recent state for which the update can be successfully applied. The runtime modifications needed to allow retrieval of appropriate previous program states are discussed in Chapter 4. We can complete the description of the system framework without referring to the details of this process.

State rollback is used to provide two guarantees prior to update application. First, it ensures that the state transformer can be successfully applied. This translates into two conditions rollback is used to enforce:

1. All instances of a class $i$ for which a state transformer $S_i$ is defined are in a consistent state. The program is not executing code contained within a synchronization block on any of these instances. However, it may be executing code within a synchronization block on objects for

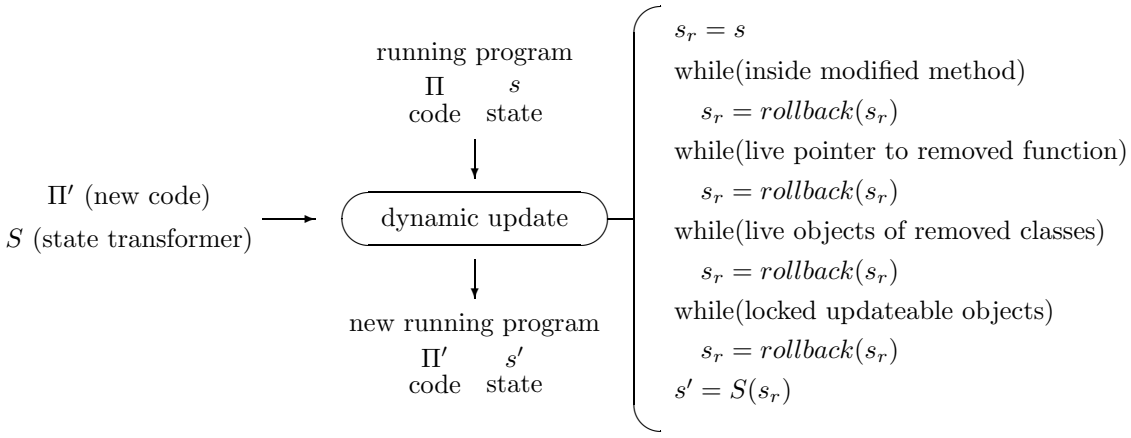



Figure 3.1: Applying an update.

which the programmer has not defined a coercion function.

2. All existing objects have a corresponding version in the new program. There are no live instances of classes which have been removed from the new program.

Second, rollback ensures there is a consistent transition point into the code of the new program. In other words, there is a semantically meaningful point at which to resume execution of the new program. This translates into an additional two conditions rollback enforces:

1. Only functions or methods that have the exact same definition in both the old and new program versions may be in the process of executing.

2. All live function pointers can be translated to corresponding references to functions in the new program version. This ensures the new program will not attempt to call a function that existed only in the old program.

Note that condition (1) above will trigger the rollback of any method the programmer modifies for the purpose of adding a synchronization block. This has the benefit of allowing the programmer to add consistency requirements after the initial program version was written. All consistency conditions in both the new and old program versions are respected by the dynamic update procedure.

## 3.3 Update

Figure 3.1 illustrates the steps involved in dynamically updating a running program. Once a running program receives an update request, its execution is halted. The runtime incrementally rolls back

the program's state until the conditions outlined in the previous section are satisfied. At this point, the programmer defined patch code is applied. The new program version resumes with the resulting state.

# Chapter 4

# Implementation

## 4.1 FJava Compiler

The Caltech Mojave compiler was a convenient architecture on which to implement our dynamic update system. It supported compilation of a type-safe, object-oriented language, FJava, which is an extension of the standard Java programming language to include higher order functions. In addition, the runtime system already featured a rollback mechanism similar to the one required for supporting dynamic updates [6].

The FJava compiler transforms a program through a number of intermediate representations in the process of producing executable machine code. The following sections outline the changes made to the compiler in order to support updateable programs.

## 4.2 Patch Specification

Patch code for transitioning between versions is developed alongside, but isolated from, the new program version. In this manner, we avoid cluttering the new program with patch code and produce a new version that can be run as a stand alone application. The patch itself is limited to (optionally) defining a state transformer $S_i$ for each class $i$ present in the new program version.

If the programmer does not specify a coercion routine for a class whose definition has changed between program versions, the update system applies a default state transformer to instances of that class. Namely, it preserves values in fields existing in both class versions and initializes new fields to a default null or zero value.

By default, only instances of classes whose definition has changed between versions are modified during the state transition process. However, the programmer may specify a coercion routine for

a class whose definition has not changed between versions in order to force the system to update instances of that class. This may be necessary if the functionality of a class changes even if its internal representation does not.

Note that in the state transformation process an object is only considered to be an instance of its true class. For example, if class $B$ is a descendant of class $A$, instances of class $B$ are affected only by the state transformer $S_B$ even though they are also instances of class $A$. Hence, $S_B$ is responsible for updating all of $B$'s inherited fields. Another reasonable choice is to call the most specific coercion function defined. Hence, if the programmer defines $S_A$ but not $S_B$, then $S_A$ is called on instances of both classes A and B. Much like calling a parent class constructor, $S_B$ should have the option of calling $S_A$ before executing itself. We do not implement these sensible alternatives simply because they introduce additional complexity without being central to the update system.

Another fine point left out of our implementation is the state transformation for static, class-level variables. In principal this is no different from coercing object instances. The programmer could define a special state transformer for each class in order to handle this aspect of an update.

Even though the new program version is independent of the patch code, they must both be available in order to update older programs. Hence, the compiler produces an executable containing both program and patch code, but enforces separation between the code bases until this last step. In particular, the compiler forbids the new program from making any calls to the patch code.

## 4.3 Intermediate Representation

### 4.3.1 Tailcalls for Speculation

As part of the compilation process, the FJava compiler transforms all programs into continuation passing form [2]. As all nonlocal jumps in control are made through the use of tailcalls, the runtime does not keep an explicit call stack. However, in order to facilitate rollback, the runtime needs access to this information. Thus, it was necessary to modify the compilation process to mark each tailcall with a tag specifying its equivalent effect in a stack-based system. For example, a tailcall corresponding to the dispatch of a method is marked differently from a tailcall corresponding to a method return. The program is still converted into continuation passing form, but the structure of the call stack is recoverable.

### 4.3.2 Name Environment

The second major modification made to the intermediate representation used by the compiler concerns the generation of symbol names. As part of the compilation process, each symbol appearing in the program or defined by the compiler is given a unique numeric name. As the program's structure does not depend on the particular choice of variable or function names, the compiler does not preserve the original names defined in the source code. However, this original name information is required in order to automatically find the correspondences between the intermediate representations of two program versions.

A significant portion of the changes to the intermediate representation focused on building an environment to keep track of all renamings made by the compiler. We created a module that could reconstruct the complete, original name for any program element, such as a function or variable definition. This original name included the type of the element, the name given by the programmer in the source code, and the scope of declaration within the source code. With this module, solving for the correspondences between two programs is reduced to finding all program elements with the same original names and types.

## 4.4 Backend and Runtime

### 4.4.1 Synchronization

Our use of the synchronized keyword did not require the complete implementation of Java synchronization locks as our system was restricted to single threaded programs. Only the main program thread could lock or unlock an object, with the imaginary updater thread only checking the status of locks. Thus, it was sufficient to add a special integer valued "lock" field to each object instance. Entering a speculation block on the object incremented this lock field, while leaving a speculation block decremented it. An object was unlocked if and only if its lock field contained a zero value.

### 4.4.2 Speculation and Rollback

In order to facilitate rollback, it is necessary to have the capability of restoring previous program states. Our runtime accomplishes this by speculatively executing methods. The runtime allocates a separate memory area on each method call. During a method's execution, a write to any program variable causes a copy of that variable, with the new value, to be created in the method's memory area. The variable's previous value is preserved in the memory area for the method from which the

current method was called. There is a stack of speculation areas with a level for each method that would appear on a traditional call stack.

Upon returning from a method, its memory area is merged into that of its caller. This does not limit rollback possibilities because any speculation blocks opened within the method would have to be closed before it exited. Hence, one a method has completed, there is never any need to rollback to a state before it has completed. Unless, of course, we need to rollback to a state before it was called. This type of rollback is still possible as the speculation level for the method's caller is still active.

Rolling back a program is the simple operation of restoring its state using a particular level of the speculation stack. By traversing the live data at a given speculation level, the specific consistency requirements stated in Chapter 3 can be checked for that level. The speculation stack also indicates the correct position at which to resume the new program - the method call that started the last valid speculation level.

# Chapter 5

# Conclusion

## 5.1 Summary

Through the use of a unique runtime system for enforcing semantic requirements, our dynamic update methodology offers significant advantages over previous work. We provide the programmer with a powerful, yet easy to use, model for designing updateable software. Our system guarantees that a dynamic update can be successfully applied to transform the state of one running program into the state of any new program in a type-safe and semantically consistent manner. The cost of this guarantee is that the running program may be rolled back to a previous state before being updated. The degree of rollback roughly reflects the amount of difference between the two program versions and should be insignificant in most practical situations.

We assure that rollback occurs only when necessary by allowing the programmer to specify exact consistency requirements. The programmer may model the dynamic update procedure as an extra thread running in parallel with the executing process. This thread is idle except when an update is requested, in which case it halts the program thread, obtains a lock on all objects scheduled for update, and calls programmer-specified patch code. Rollback of the program thread's state occurs when the updater thread is not able to obtain all of the desired locks. From the programmer's perspective, consistency of an object's internal data is enforced just as it would be in a multithreaded program. The programmer takes a synchronization lock on an object when beginning critical manipulations of that object's internal data and releases the lock when finished.

A second condition necessitating rollback guarantees a correct transition point into the new program's code once an update has completed. In particular, the system rolls back the old program so that no method modified in the new program version is live at the time of the state change. This ensures that only code present in the new program version will be executed after an update

completes. It also ensures that the update process respects any consistency requirements introduced in the new version, but not present in the old version. Using this feature, the programmer can build a semantically consistent update for a program not originally designed to be updateable.

In order to support state rollback, the runtime system speculatively executes methods. The runtime maintains a copy of the program state at the beginning of each method on the execution stack. During the execution of a particular method, both the space and time overhead associated with the speculation mechanism are proportional to the amount of data modified by that method. With this minimal cost, our system provides robust guarantees on updateablity not offered by any previous dynamic update mechanism.

## 5.2 Future Work

In our current implementation, each call to or return from a method incurs overhead as a result of the need to manage the speculation stack. In rare cases, this overhead may be unacceptable for the desired performance of the program. As the sole purpose of speculating on each method dispatch is to reduce to a minimum the rollback associated with any future update, much of this speculation may be unnecessary. In particular, the programmer may have an idea of how future updates will work, restricting the methods for which speculation is appropriate. Alternatively, the programmer may opt to pay a larger rollback cost at update time for the efficiency gained by not speculating on some method calls. This could be a sensible choice for algorithms which would otherwise incur significant overhead on the speculation stack.

A simple extension to the current implementation could provide the programmer with the power to appropriately balance speculation overhead with update rollback costs. We could add a keyword to the programming language which, when placed before a method, designates that speculation should not occur on calls to that method. Finer control over the runtime costs of updateable code could extend the desirability of our system to critical applications in which performance is also a key factor.

A similarly important task is to extend our current framework to support updateable multi-threaded applications. For such applications, an update system faces the additional challenge of guaranteeing the global consistency of the states of each thread at the time of the update. For example, if thread A waits and is eventually released by thread B, then whenever thread B is rolled back to a point before the release, thread A should be rolled back to the point at which the wait began. Similarly, if thread A spawns thread B, then whenever thread A is rolled back to a state prior

to this action, thread B must be rolled back to the state of nonexistence. The challenge of supporting multithreaded applications centers on creating a speculation mechanism capable of tracking such dependencies with minimal runtime overhead. A framework for handling multithreaded programs could significantly enhance the applicability of our dynamic update system to real world problems.

# Bibliography

[1] U. Acar, G. Blelloch, R. Harper. *Adaptive Functional Programming.* ACM Symposium on Principles of Programming Languages, January 2002.

[2] A. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[3] O. Frieder and M. Segal. *On Dynamically Updating a Computer Program: From Concept to Prototype.* The Journal of Systems and Software, pp. 111-127, February 1991.

[4] D. Gupta. *On-line Software Version Change.* Ph.D. Thesis, Department of Computer Science and Engineering, Indian Institute of Technology, Kanpur, 1994.

[5] D. Gupta, P. Jalote, and G. Barua. *A Formal Framework for On-line Software Version Change.* Transactions on Software Engineering, 22(2):120-131, February 1996.

[6] J. Hickey, J. D. Smith, B. Aydemir, N. Gray, A. Granicz, and C. Tapus. *Process Migration and Transactions Using a Novel Intermediate Language.* California Institute of Technology Technical Report caltechCSTR:2002.007. 2002.

[7] J. Hickey and others. *Mojave Research Project Home Page.* http://mojave.caltech.edu/

[8] M. Hicks. *Dynamic Software Updating.* Ph.D. Thesis, Computer and Information Science, University of Pennsylvania, Philadelphia, PA, 2001.

[9] M. Hicks, J. T. Moore, and S. Nettles. *Dynamic Software Updating.* Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI'01), June 2001.

[10] G. Hjalmtysson and R. Gray. *Dynamic C++ Classes: A Lightweight Mechanism to Update Code in a Running Program.* Proc. of the USENIX Annual Technical Conference, New Orleans, LA, June 1998.

[11] I. Lee. *DYMOS: A Dynamic Modification System.* Ph.D. Thesis, University of Wisconsin, 1983.

[12] S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J. F. Barnes *Runtime Support for Type-Safe Dynamic Java Classes.* Proceedings of the Fourteenth European Conference on Object-Oriented Programming, June 2000.

[13] K. Sunil. *Design and Implementation of an On-line Software Version Change System for Objective-C.* Master Thesis, Department of Computer Science and Engineering, Indian Institute of Technology, Kanpur, 1997.

[14] Z. Tang. *Dynamic Reconfiguration of Component-Based Applications in Java.* Master Thesis, Computer Science and Engineering, Massachusetts Institute of Technology, Cambridge, MA, September 2000.