Dynamic Software Updates: The State Mapping Problem *

Rida A. Bazzi Kristis Makris Peyman Nayeri Jun Shen
Arizona State University
{bazzi,kristis.makris, peyman.nayeri, jun.shen.1}@asu.edu

Abstract

We consider the state mapping problem for dynamic software updates and propose a number of approaches that have the potential of automating the state mapping in practical setting.

Categories and Subject Descriptors D.2.7 [*Software*]: Distribution, Maintenance, and Enhancement—Enhancement

General Terms Live Software Update, Program Slicing, Unification, Bug Fixes

Keywords Software Evolution, Dynamic Updates, Backward Compatibility

1. Introduction

The dynamic software update (DSU) problem consists of two components. First, there is the need to determine if it is *meaningful* (safe) to map a state of the old application to a state of the new application, and, if it is, determine the required mapping - this is the *state mapping problem.* Second the state mapping needs to be effected through a mechanism that maps an old execution state to a new execution state - this is the *update mechanism problem.* A general update mechanism was proposed in [1]. The mechanism supports a general state mapping for multithreaded applications including the mapping of active functions state while preserving system resources used by the application(file descriptors, and socket connections in particular). This paper does not address the update mechanism problem.

The state mapping problem is undecidable [2]. This implies that, in general, user help is needed to determine safe update points and to specify the state mapping function. Nevertheless, this does not mean that it is not possible to solve the state-mapping problem automatically or semi-automatically without or with little user help for many practical cases of interest. Proposing approaches that bring us closer to achieving practical state mapping is the goal of this paper.

* This material is based upon work supported in part by the National Science Foundation grant #0849980.

HotSWUp'09, October 25, 2009, Orlando, Florida, USA.
Copyright © 2009 ACM ISBN 978-1-60558-723-3/09/10. . . $10.00

```
 ...                       ...
                          if (x > y)
x = y+z;                     x = y+z;
w = x/2;                   w = x/2;
x = w-z;                   x = w-z;
/* update point */

 (a) old version           (b) new version
```

Figure 1. The bug fix problem

The approaches we propose are pragmatic. They vary between ignoring some differences between the two versions because they cannot be addressed (bug fixes and corrupted states) or are benign (changes to log functions), reducing the amount of state that needs to be mapped (waiting for light weight functions to exit) or working hard to ensure a correct mapping for backward compatible features.

2. Mapping the state

One can identify the following general categories of software updates: (1) bug fixes; (2) performance improvement; or (3) adding features (this list is not meant to be comprehensive). These categories affect how we look at the state mapping problem (for lack of space, we do not consider performance improvement).

Some amount of backward compatibility is implicitly assumed by dynamic software update systems. Otherwise, dynamic update does not make much sense: in order to produce a state of the new version from a state of an old version, whether with user help or automatically, the two states must be somewhat related. In this paper we are interested in the problem of automatically mapping all or a portion of the state from one version to another. This problem is related to the compatibility assumptions.

2.1 Bug Fixes: the checkpointing approach

A software update that aims at fixing a bug is essentially incompatible with the version it aims at fixing. One can have one of two positions vis à vis such an update. One the one hand, one can consider that it is beyond the scope of the DSU state mapping problem to deal with such a situation. The reason is that the state to be mapped might be corrupted and detecting and fixing a corrupted state is not part of the DSU problem. A variation on this position is to assume that the state is not corrupted and hope for the best! One the other hand, one can consider that mapping a corrupted state to a non-corrupted state is the quintessential state mapping problem. The goal is to detect the corruption

```
...                         ...
                            if (newOption)
                             x = y*z;
                            else
                               x = y+z;
x = y+z;                     w = x/2;
w = x/2;                     x = w-z;
x = w-z;
/* update point */

(a) old version              (b) new version
```

Figure 2. Code Enhacement

in the state, if any, and replace the state with a non-corrupted state. The code fragment in Figure 1 illustrates some of the issues in this case.

In the code, x is incorrectly calculated if $x \leq y$. At the update point this information is not available because x has been redefined. Even though in this program it is possible to execute the statement in reverse to determine the value of x at the first assignment, in general this is not possible, but that is exactly what is needed. This can be achieved with checkpointing. An extreme, and impractical, approach is to checkpoint after every statement. A somewhat more practical approach is to checkpoint at function entries and log all interactions with the environment and then re-execute functions from their entry points at the time of an update. The re-execution would only simulate interaction with the environment and compare the new interaction with the old ones for consistency. If the interactions with the environment are consistent with the old ones, then the new internal state can be *computed*. In the example above, the corruption of the state can be determined if the value of x between the two versions differ.

2.2 Enhancements: slicing with unification

Enhancements are an important category of updates that add features to existing applications. The new application can have new state components that implement new options. The state mapping in this case should be done in such a way that preserves the old execution. In other words, the execution after the update would be equivalent (when restricting attention to part of the new state) to the old execution. The problem is to determine how the new state could be set to that effect. Figure 2 illustrates this situation. In the figure, the state has a new variable *newOption* that controls how the execution proceeds. If *newOption* is **true**, then new behavior is produced, otherwise the old behavior is maintained. So, at the update point *newOption* should be set to **false** and a general approach is needed to determine that.

Existing works that compare the behavior of programs [3] aim at establishing that two programs or pieces of code are identical. The problem we are facing is related, but with a twist. The problem is: given two programs, how can you assign values to free variables so that the program have identical behavior. Also, the definition of identical has to be relaxed to one of compatible. For example an integer variable is compatible with a long variable as far as update is concerned (but not the other way around). This recalls work that uses unification and that was used successfully for type systems, but also for other problems such as alias analysis [4, 5].

We propose an approach whereby unification is combined with techniques to analyze behavioral equivalence (program slicing and dependency graphs comparison) to achieve a mapping that results in an execution of the updated program that is compatible with the original execution. Such an analysis need not always produce positive results (attempting to unify changes due to bug fixes would not work because there are no *free* (new) variables in such cases). Exploring this approach is an exciting line of research.

2.3 Smaller state: light-weight functions

In order to avoid mapping active functions on the stack, DSU systems have traditionally opted to wait for quiscent points. This also has the advantage of mapping a smaller state. The update mechanism of [1] can map active functions and in general one cannot wait for a quiescent point. We adopt a middle ground. An analysis (conservative) of the program can come up with a list of functions that can be expected to exit in a timely fashion. We call these functions *light-weight functions*. The update system will ensure that no update points are in light-weight functions. This way, there is no obligation to handle the mapping of the state of these functions. A preliminary (and primitive) implementation of program analysis for light-weight functions indicate that at least 20% and up to 60% of functions can be easily determined to be light-weight. We expect that further analysis will produce better numbers.

2.4 Ignoring some changes: log functions

Some changes to the code can be considered benign. For example, the format of error messages produced by an application might change. If such messages are consumed outside the application whether by a human or by other applications that read log files (and that are not available to the update system), then a reasonable approach is to ignore such changes in format as harmless. This would require that we detect changes that are log-related. We have already implemented some heuristics to detect log-related changes. The heuristics take into consideration the function signature, the function name and the calling pattern to determine if a function is log-related. Initial results for a number of real-world applications indicate that 10% of changes can be attributed to automatically detected log functions.

3. Conclusion

We believe that it is possible to address the general state mapping problem by adopting a pragmatic approach that aims at identifying cases that restrict the problem or by eliminating some cases from consideration.

References

[1] Kristis Makris and Rida A. Bazzi. Immediate Multi-Threaded Dynamic Software Updates Using Stack Reconstruction. *U*SENIX'09 Annual Technical Conference, 2009.

[2] Deepak Gupta, Pankaj Jalote, and Gautam Barua. A formal framework for on-line software version change. *Software Engineering*, 22(2):120–131, 1996.

[3] Wuu Yang, Susan Horwitz, and Thomas Reps. Detecting program components with equivalent behaviors. Technical Report CS-TR-1989-840, 1989

[4] Robin Milner. A Theory of Type Polymorphism in Programming. *J.* Comput. Syst. Sci. 17(3): 348–375, 1978.

[5] Manuvir Das. Unification-based pointer analysis with directional assignments. *P*LDI '00: Programming language design and implementation, 35–46, 2000.