Synthesizing Object Transformation for Dynamic Software Updating

Tianxiao Gu, Xiaoxing Ma, Chang Xu, Yanyan Jiang, Chun Cao, Jian Lü,
State Key Laboratory for Novel Software Technology, Nanjing University, China
{tianxiao.gu, jiangyy}@outlook.com, {xxm, changxu, caochun, lj}@nju.edu.cn

***Abstract*—Dynamic software updating (DSU) can upgrade a running program on-the-fly by directly replacing the in-memory code and reusing existing runtime state (*e.g.*, heap objects) for the updated execution. Additionally, it is usually necessary to transform the runtime state into a proper new state to avoid inconsistencies that arise during runtime states reuse among different versions of a program. However, such transformations mostly require human efforts, which is time-consuming and error-prone. This paper presents AOTES, an approach to automating object transformations for dynamic updating of Java programs. AOTES tries to generate the new state by re-executing a method invocation history and leverages symbolic execution to synthesize the history from the current object state without any recording. We evaluated AOTES on software updates taken from Apache Tomcat, Apache FTP Server and Apache SSHD Server. Experimental results show that AOTES successfully handled 47 of 57 object transformations of 18 updated classes, while two state-of-the-art approaches only handled 11 and 6 of 57, respectively.**

I. Introduction

Traditional software updating techniques need to shutdown the running software systems, which may lead to unpredictable losses. Dynamic software updating (DSU) can eliminate these losses by updating running software systems without stopping them. Specifically, a typical DSU system replaces the existing in-memory code by its new version and then reuses existing runtime data for the execution of the new version of code.

Actually, replacing the code on-the-fly is not very difficult due to runtime code manipulation facilities such as dynamic linking [1] and dynamic class loading [2]. The main challenge is to handle inconsistencies that arise during data reuse among different versions of a program. Existing DSU systems usually apply a transformation to runtime data that may give rise to inconsistency errors.

In theory, runtime state transformations for arbitrary programs cannot be *fully* automated [3]. Most DSU systems attempt to automatically generate a transformation based on some predefined rules. Among them, the most popular approach is *default transformation* [4], [5], [6]. Specifically, a default transformation preserves the value of an unchanged field and assigns a *type-specific default value* (*e.g.*, 0 for `int`) to a newly added field in an updated object. Obviously, default transformations may inevitably fail. Therefore, existing DSU systems also provide interfaces for developing transformation functions (*i.e.*, *transformers*) [4], [5], [6]. However, developing transformers is time-consuming and error-pone. Developers need to capture the correspondence as well as the difference between the quick-change runtime states of the old and new versions of a program.

Target object synthesis (TOS) [7] tries to apply a set of rules to learn a transformer by examples. A transformation example is two objects selected at some corresponding point during the execution of a same test over the old and new version, respectively. However, TOS may easily fail to generate a transformer even when it has collected sufficient examples. This is mainly because objects are usually implemented by complicated data structures and primitive rules may not be adequate to handle unpredictable changes of such implementations.

To address the problem, we try to avoid manipulating the low level implementation of objects. We observed that first the current state of an object is indeed produced by the execution of the method invocation history on the object and second the method invocation history of an object is usually unchanged during dynamic updating. Therefore, the new state can be created by re-executing the same method invocation history under the new version of methods on the new initial state.

Take an array based container as an example. The method invocations `add(a)`, `add(b)` and `remove(a)` on an empty container fills the container with the only element `b`. Suppose that we update the array based implementation to a linked list based one. We can easily see that the new state can be created by re-executing the same history on an empty linked list based container.

Note that a method invocation history contains not only methods but also all their arguments. A record-and-replay approach is prohibitively expensive. First, logging every method invocation significantly slows down the execution and also needs tremendous space for storing log data. Second, an actual history may contain irrelevant method invocations (*e.g.*, `add(a)` and `remove(a)`). Replaying a long history with such method invocations may cause a long service interruption.

To this end, we try to synthesize an *equivalent* method invocation history only based on the current state of an object. An equivalent method invocation history can lead to the same object state from the initial state as the actual history but may be more compact in terms of fewer redundant method invocations. For the previous example, only a single method invocation `add(b)` is equivalent to the whole actual history.

However, synthesizing a method invocation history is also non-trivial. The main challenge comes from providing a proper value for a parameter. Besides, the online synthesis has a strict

time limit, which prevents us from using a time-consuming search algorithm such as backtracking.

To alleviate these difficulties, we try to synthesize a specific kind of *inverse methods* instead of the original methods for history synthesis. An inverse method takes *no* argument and can revert the current state back to a previous state before invoking the corresponding original method. Thus, we can synthesize an inverse invocation history that produces the initial state back from the current state using a greedy algorithm and then revert the inverse method invocation history.

II. Approach

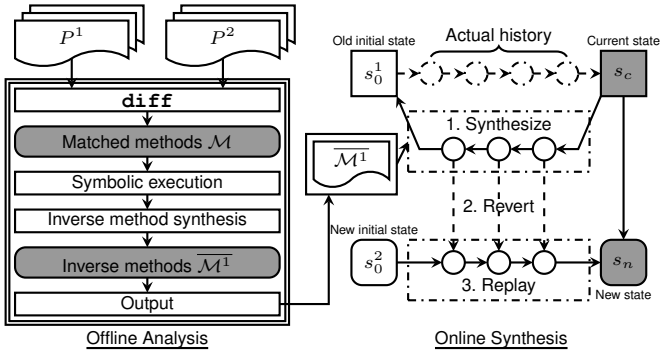

Fig. 1: Approach overview.

Figure 1 presents an overview of AOTES. AOTES works in two phases, *i.e.*, the offline analysis and the online synthesis. During the offline analysis, AOTES takes as input only two versions of a Java program's byte code and outputs a group of inverse methods for *matched methods* of changed classes only. Here, we match methods by their names and signatures. Apparently, it is hard to synthesize a *complete* inverse method covering all execution paths for every original method. AOTES only considers a set of *short* execution paths for an original method and synthesizes an inverse method for each of them.

During dynamic updating, AOTES takes over the responsibility to initialize the new state for a stale object from the underlying DSU system. After receiving the stale object for transforming, AOTES tries to search for an inverse method invocation history, reverts it, and replays the reverted history on the new initial state to initialize the updated new state.

III. Evaluation

We collected 18 updated classes from Apache FTP Server, Apache SSHD Server and Apache Tomcat, which are all widely used server applications under years of active development. We classified these changes into the following types:

1) Value Change: with no field added, but the values of some fields need to be updated.
2) Name Change: with a field renamed only[1].
3) Type Change: with a type-changed field only.
4) Other Change: any other changes.

[1]Note that if either the name or type of a field is changed, it is considered as deleted and a new field with the new name or type is added.

We manually created 57 tests in total for validation. Every test created an object with a few method invocations before dynamic updating. After that, we triggered the dynamic updating and applied the transformation to the object. Every dynamic updating was verified as follows. That is, the state after dynamic updating of the old version must be equivalent to a state that can be achieved by executing the same methods on the new version. We ran all tests with dynamic updating on Javelus [8], [6] for AOTES and default transformation, respectively. All results are shown in Table I. AOTES succeeded in 47 (82.5%) updates and failed in other 10 updates due to *incomplete* or *inconsistent* synthesized histories.

TABLE I: Results of real-world updates.

Type	Update	Tests	AOTES	Default	TOS
Value Change	tomcat-dd741c	6	4^{α}	4	N.A.
Name Change	ftp-5d5592	4	3^{α}	0	N.A.
	sshd-6f8507	2	2	1	N.A.
	tomcat-9510e8	2	2	1	N.A.
	tomcat-b75f5c	2	2	1	N.A.
Type Change	ftp-f8110b	5	5	0	N.A.
	sshd-054334	3	3	1	N.A.
	tomcat-480edc	3	3	0	N.A.
Other Change	ftp-43ff5f	4	2^{β}	0	N.A.
	ftp-4907aa	4	2^{β}	0	N.A.
	ftp-5df186	4	2^{β}	0	N.A.
	sshd-009d83	5	5	0	N.A.
	sshd-1487b0	1	1	0	1
	sshd-2297b2	1	1	0	1
	sshd-b98694	7	7	2	2
	sshd-eeeec6	1	0^{α}	0	1
	tomcat-24bc4d	2	2	1	1
	tomcat-2db0f7	1	1	0	N.A
	Total	57	47/82.5%	11/19.3%	6/10.5%

α and β indicates inconsistent and incomplete synthesized history, respectively.

An incomplete history cannot properly initialize all fields. This is mainly because many native methods prevented AOTES from generating sufficient inverse methods. We plan to support more native methods in future. An inconsistent history produces the same state as the actual history in the old version but a different state in the new version. Actually, our manually created validation tests are too strict. Because both the synthesized and the actual history can produce the same current state, even a developer cannot prepare a transformation that fulfills the validation tests for these updates without information beyond the current state (*e.g.*, the actual history).

We did not run TOS with dynamic updating as TOS is not fully automated and requires extra training tests and manually specified update points. Instead, we used our validation tests to train TOS. TOS failed to synthesize a transformer for 13 of 18 updates (marked with N.A. in Table I). For the rest 5 updates with 12 tests in total, TOS even failed in validating its transformer against 6 training tests.

Acknowledgements

This work was supported by the 973 Program (No. 2015CB3 52202) and NSFC (Nos. 61690204, 61472177) of China.

REFERENCES

[1] R. C. Daley and J. B. Dennis, "Virtual memory, processes, and sharing in MULTICS," *Communications of the ACM*, vol. 11, no. 5, pp. 306–312, 1968.

[2] S. Liang and G. Bracha, "Dynamic class loading in the Java virtual machine," in *Proceedings of the ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, 1998, pp. 36–44.

[3] D. Gupta, P. Jalote, and G. Barua, "A formal framework for on-line software version change," *IEEE Transactions on Software Engineering*, vol. 22, no. 2, pp. 120–131, 1996.

[4] S. Subramanian, M. Hicks, and K. S. McKinley, "Dynamic software updates: A VM-centric approach," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009, pp. 1–12.

[5] T. Würthinger, C. Wimmer, and L. Stadler, "Dynamic code evolution for Java," in *Proceedings of the International Conference on the Principles and Practice of Programming in Java*, 2010, pp. 10–19.

[6] T. Gu, C. Cao, C. Xu, X. Ma, L. Zhang, and J. Lü, "Low-disruptive dynamic updating of Java applications," *Information and Software Technology*, vol. 56, no. 9, pp. 1086–1098, 2014.

[7] S. Magill, M. Hicks, S. Subramanian, and K. S. McKinley, "Automating object transformations for dynamic software updating," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 2012, pp. 265–280.

[8] T. Gu, C. Cao, C. Xu, X. Ma, L. Zhang, and J. Lu, "Javelus: A low disruptive approach to dynamic software updates," in *Proceedings of 19th the Asia-Pacific Software Engineering Conference*, 2012, pp. 527–536.