Automated Recommendation of Dynamic Software Update Points: An Exploratory Study

Zelin Zhao, Xiaoxing Ma, Chang Xu, Wenhua Yang
State Key Laboratory for Novel Software Technology at Nanjing University
Institute of Computer Software, Nanjing University
{zelinzhao1105, ihope1024}@gmail.com, {xxm, changxu}@nju.edu.cn

ABSTRACT

Due to the demand for bugs fixing and feature enhancements, developers inevitably need to update in-use software systems. Instead of shutting down a running system before updating, it is often desirable and sometimes mandatory to patch the running software system on the fly, with a mechanism generally referred as *dynamic software updating* (DSU). Practical DSU strategies often require manual specification of *update points* in the program for performing dynamic updates. At these points DSU systems will update the program code, and also migrate the program state to the new version program (using *transformation functions*). However, finding appropriate update points is non-trivial because the choice of update points has great influence on two competing factors: the timeliness of DSU and the complexity of transformation functions; and to strike a good balance between them requires a deep understanding of both versions of the program. In this exploratory paper, we conceive an automated approach to the recommendation of update points for developers. We conduct a set of preliminary experiments with a real world software update case to examine the feasibility of the approach.

Categories and Subject Descriptors

B.5.2 [Design Aids]: Automatic synthesis; D.2.7 [Distribution, Maintenance, and Enhancement]: Maintenance;

General Terms

Design, Languages, Experimentation.

Keywords

Dynamic software updating (DSU); update points; automated recommendation

1. INTRODUCTION

In use software systems often need to be updated to correct faults, improve performance and add functionalities. General software update schemas require that the to-be-updated system is static. If the system is running, one needs to terminate it first, update it and then restart it. However, in some situations, the stop-and-restart update schema is undesirable and even unacceptable. For example, stopping an air traffic controlling system or a life supporting system can cause life loses, offlining a financial transaction processing system can bring money loses. Even for daily used software systems, frequent restarts for applying patches can be disruptive and annoying.

Dynamic software updating (DSU) is a generic technique that can alleviate these problems by updating software systems at run-time. DSU systems apply *dynamic patches* [1,8,9,14] on running programs. In addition to the code changes to the program, a dynamic patch also specifies a set of *update points* and corresponding *transformation functions*. Update points are positions in the original program where the program states are "steady" and can be reasonably mapped onto appropriate states of the new program, from which the program can continue in the new version. Transformation functions define how to do the mapping.

A well prepared dynamic patch should make the DSU process timely and safe, i.e., the update can be applied as soon as possible and the new version can execute as expected after the update. The selection of dynamic update points has great influence on the timeliness of DSU and the complexity of transformation functions. On one hand, since one cannot predict when the dynamic updated request is raised at run-time, the more update points are allowed, the timelier the DSU can be. On the other hand, it can be more difficult to develop correct transformation functions for all these update points.

Some existing DSU approaches such as Jvolve [1] and Javelus [14] implicitly specify update points by excluding infeasible points. These approaches apply dynamic updates once the program is at DSU *safe points* [1]. DSU safe points are special VM safe points where no modified methods are currently active. By allowing all DSU safe points as update points, the timeliness of DSU can be very good in most cases. However, developers need to prepare an universal transformation function that can map any possible state at these safe points. This task, if possible, can be very difficult except for those trivial updates.

Hayden et al. suggest that selecting update points manually is the more effective way [2]. However, one have to thoroughly understand the logic and runtime state evolution of both version, and then select several update points from a huge amount of program points, which is time-consuming and labor-intensive.

We propose to alleviate this program by automatically recommend a small set of candidate points for developers. The idea is based on following observations. First, dynamic software updates are

INTERNETWARE'14, November 17, 2014, Hong Kong, China
Copyright 2014 ACM 978-1-4503-3303-0/14/11...$15.00
http://dx.doi.org/10.1145/2677832.2677853

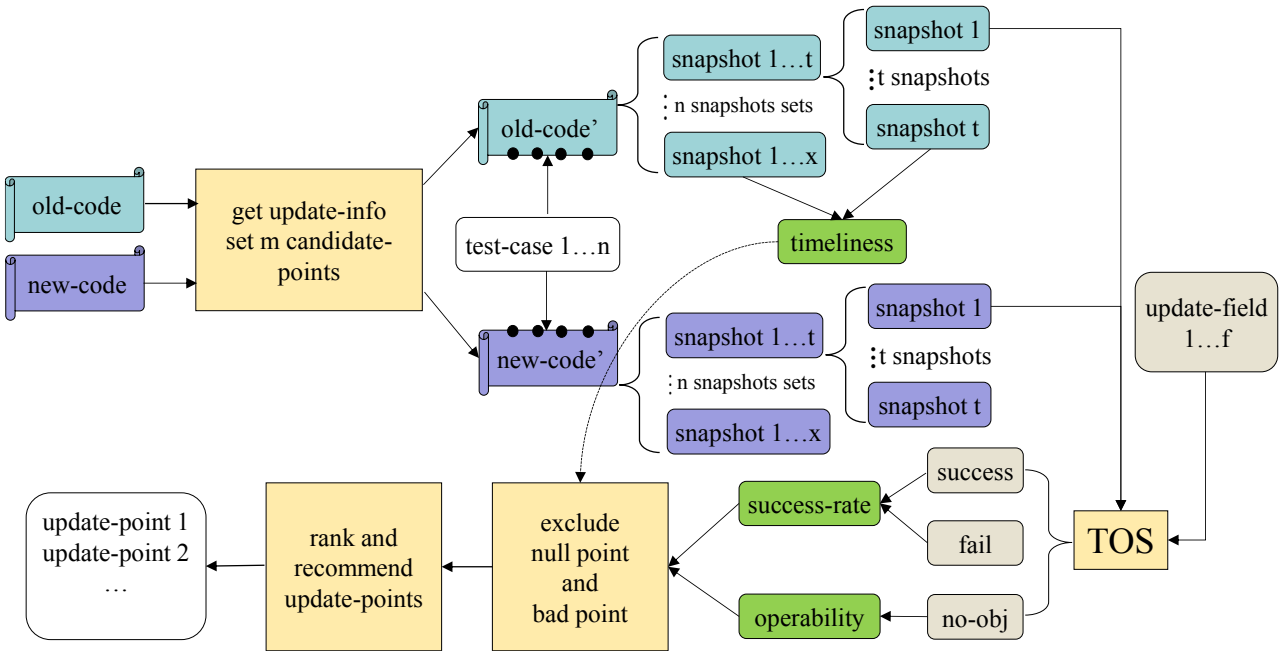

Figure 1. Automated recommendation of update points

often evolutionary rather than revolutionary, otherwise it would be meaningless to apply it dynamically. In addition, for evolutionary changes the most of test cases provided along with the software are same for both versions. Second, there is a strong correspondence between the runtime states at corresponding positions in the two versions, and this correspondence can be observed by monitoring the executions of both version under same test inputs. TOS [4] exploited this correspondence to synthesis transformation functions for given update points, but we plan to further leverage it to recommend those program points that demonstrates clear state correspondence and are frequently passed through.

The rest of this paper is organized as follows. Section 2 introduces the background and our problem setting. Section 3 defines some properties of update points and describe our conceived approach in details. Section 4 overviews an exploratory study for the approach, and Section 5 gives the experiment results. Before concluding the paper in Section 7, we discuss some weaknesses of the approach and the experiments in Section 6.

2. BACKGROUND AND PROBLEM SETTING

In this section we first briefly discuss how DSU works, then introduce the TOS tool that can automatically generate transformation functions, and finally explain our problem setting.

2.1 Dynamic Software Updating

Once a dynamic update request is received during the run-time of a program, the DSU system will trap the program at a dynamic update point, update the code and state of the program, and then resume the execution with the new code. Updating the code image of a running program is not difficult with the support of modern operating system and/or programing language runtime. However, in order to run the new version program correctly, DSU systems also need to migrate the state of the old version program to new version program and make it compatible with new version program. Transformation functions are called to do this work. Some DSU systems, like Jvolve [1] and Javelus [14] can produce default transformation functions automatically, by analyzing the bytecode of old and new programs. However, automatically generated transformation functions can only guarantee type safety but not semantic correctness of following program execution. For example, they simply assigning default value to new fields (e.g. **0** for **int** and **null** for **String**).

In practice, these changed fields often need specific values instead of default values. For example, if we change the definition of a field named *EmailAddress*, which should save the email address, in an email server system. The default transformation functions would assign **null** to *EmailAddress*. But we all know that *EmailAddress* in the new version program should save the email address, which we could get from the old version program. Consequently, while we use Jvolve or Javelus to update a program dynamically, we need to modify the default transformation functions sometimes.

2.2 Targeted Object Synthesis

Given human selected update points and a set of test cases common for both versions of the program, Targeted Object Synthesis (TOS) [4] can automatically produce transformation functions for updated fields. TOS extracts old and new objects of updated classes from old and new memory snapshots separately, then analyzes objects to produce transformation functions. The process is divided into two phases, MATCH and SYNTHESIS. TOS matches old and new objects up in MATCH phase and passes these pairs-of-objects to SYNTHESIS. In SYNTHESIS, TOS analyzes the values of each field in each pair of objects and synthesizes transformation functions for each field.

Each time TOS tries to generate a transformation function for an updated field, there would be three kinds of results: generated successfully, unsuccessfully or there are no objects of the updated

class at this program point. We consider the first result as *success*, the second as *fail* and the third as *no-obj*.

2.3 Selecting Dynamic Update Points

As mentioned before, allowing dynamic update happen at any DSU safe point could make the transformation functions very hard to define. Manual selection of update points works in practice but requires deep understanding of both versions of the software and can be labor-intensive. One way to simplify the task is to automatically recommend a small set of candidate update points for developers.

More elaborately, given two versions of program, a set of common test cases for the two versions, we want to automatically find a set of candidate update points that satisfy the following requirements:

(1) *Timeliness*. Once a dynamic update requests is raised, an update point should be reached in a short time. This implies that the execution of program pass through these update points frequently.

(2) *Correctness*. The program should behave correctly after the update. Since complete formal specifications are seldom available in practice, we just require for all the provided test cases the program behave as expected, despite of the dynamic updates.

The selected candidates of update points, together with the transformation functions generated by existing tools such as TOS, are recommended to developers for final consideration.

3. AUTOMATED RECOMMENDATION OF UPDATE POINTS

3.1 Observable Properties of Update Points

Before presenting our conceived approach, we define three observable properties of update point. In our approach, we evaluate and recommend update points based on these properties.

Definition 1. *Timeliness* means the frequency of passing through a candidate point during the execution of the program.

The higher this frequency is, the more opportunities that dynamic update can be updated at run-time.

Definition 2. *Success-rate* means the proportion of successful generation of transformation functions in all tries for updated fields at a candidate point.

When we apply the update, we need to transform old objects to new ones compatible with new version program. Assume there are *c* updated classes and they have *f* updated fields. The *c* updated classes all have their own objects at an update point. If we apply update at this update point, we need *f* transformation functions to migrate each field in each updated object. In our approach, we need an automated tool (e.g. TOS [4]) to generate transformation functions for updated fields. So the success-rate indicates the level of difficulty of producing transformation functions.

Definition 3. *Operability* means the proportion of updated class which have no objects at a candidate point.

If the number of updated classes with live objects is little at a candidate point, we just need to generate transformation functions for few updated fields at this candidate point. In extreme cases, none of the updated classes have live objects at this point (*operability* is 1), which means that no objects need to be transformed during the progress of updating, and we do not need to generate transformation functions.

3.2 A Conceived Approach

As we mentioned before, default transformation functions are not satisfied with the practical needs and we should generate transformation functions according the program state at update points. That means the selection of update points has a great influence on the difficulty of generating transformation functions. More important, update points directly affect whether to apply the update successfully. Therefore, we must determine update points cautiously.

Selecting update points manually is a very effective way. Unfortunately, the cases with manual intervention are time-consuming and usually challenging to be right.

We conceive an automated recommendation of update points, shown in Figure 1, to help programmers select update points. Given source code of the old and new versions and some test cases, our approach can evaluate candidate points and recommend some update points for programmers. Our approach is suitable for the situation which meet the following conditions: 1) The test cases is sufficiently representative and 2) the specification of the program is not changed or 3) the specification is changed a little but the changed part hasn't been reached yet. For now, we implemented our approach for Java program.

Figure 1 shows the steps of our approach. First, we need to prepare an old version source code *old-code* and a new version source code *new-code*. Also, we need to get *n* test cases, which can execute on *old-code* and *new-code*. By analyzing the source code, we can obtain information about classes, fields and methods (including constructors). Then we compare the source code information of both versions to get the update information, which contains updated classes, fields and methods. We also record the information of unchanged methods. Update points should be located in unchanged methods, so we set *m* candidate points in unchanged methods by inserting a little piece of Java code in almost each line of effective code. Afterwards, we get *old-code'* and *new-code'*, and run the same *n* test cases on *the old-code'* and *new-code'* respectively. Running one test case on *old-code'* or *new-code'* will pass by some candidate points one or more times. We capture a snapshot at each time reaching a candidate point. The memory snapshot contains all live objects which are created during the execution of Java programs.

We write a Java program named *Catch.java* which has a method named *snapshot*. The parameter of *snapshot* is **int** *PointNo*, which indicate the identifier of candidate points. In *snapshot*, we call java.lang.Runtime to execute Jps [16] and Jmap [16] to dump snapshots. If we want to set a candidate point in one line of code, we just need to insert *Catch.snapshot(PointNo)* behind this line. Because the effective code of unchanged methods in old-code and new-code are exactly same, we should get exactly the same results after we set candidate points in unchanged methods. In Table 1, there is an unchanged method *f()* in *old-code* and *new-code* before setting candidate points and they are exactly the same. The results of setting candidate points (assuming the *PointNo* are 10 and 11) in *f()* are shown in Table 2 and they are exactly the same too.

While executing $TestCase_i$ on the *old-code'*, the *c*th time of calling *f()*, *Catch.snapshot()* will dump 2 snapshots (*old-snapshot10-c* and *old-snapshot11-c*). The same happens while executing $TestCase_i$ on *new-code'* (dumping snapshots *new-snapshot10-c* and *new-snapshot11-c*).

Table 1. Unchanged methods without candidate points

old-code	new-code

`void f() {`
`  int num = 1;`
`  String str = null;`
`}`
	`void f() {`
	`  int num = 1;`
	`  String str = null;`
	`}`

Table 2. Unchanged methods with candidate points

old-code'

`void f() {`
`  int num = 1; Catch.snapshot(10);`
`  String str = null; Catch.snapshot(11);`
`}`

new-code'

`void f() {`
`  int num = 1; Catch.snapshot(10);`
`  String str = null; Catch.snapshot(11);`
`}`

Afterwards, we can make *old-snapshot10-c* and *new-snapshot10-c* as a pair, *old-snapshot11-c* and *new-snapshot11-c* as another pair.

Running the *n* test cases on each version program produces *n* collections of snapshot and each collection contains some snapshots. We consider the two collections, produced by running same test case on both versions of the program, as corresponding collection. We detect whether the corresponding collections contain the same number snapshots. If not, we will eliminate these redundant snapshots. In our experiment, most corresponding collections have the same number snapshots. We match the snapshots in corresponding collections one by one.

After all test cases are performed on *old-code'*, we can calculate the *timeliness* of each candidate point. We count the total number (*total-count*) of snapshots in all collections of snapshot. Meanwhile, we count the number (*candidate-count*) of snapshots belong to each candidate point. The *timeliness* of candidate point is the ratio of *candidate-count* and *total-count*.

$$timeliness = candidate\text{-}count / total\text{-}count$$

After collecting all the snapshots of both *old-code'* and *new-code'* and pairing them up, we will invoke TOS [4]. TOS automatically produce transformation functions by analyzing a pair of old and new snapshots. To explain it, we take the first collection of snapshots as an example and we assume there are *t* snapshots in it. Because we've matched the snapshots up, the snapshot *i* in old version and snapshot *i* in new version are pairs-of-snapshots. We use TOS to analyze each pair-of-snapshots, trying to produce transformation functions for *f updated-fields*. We treat producing transformation functions successfully as *success*, unsuccessfully as *fail*. Moreover, there may be no object of the updated class in one pair-of-snapshots, and we consider this result as *no-obj*. After we finish generating transformation functions for updated fields, we count the number of each result, *success-count* for *success*, *fail-count* for *fail* and *no-count* for *no-object*. The sum of *success-count, fail-count* and *no-count* should equal to *f*. The *success-rate* of one pair-of-snapshots is the ratio of *success-count* and the sum of *success-count* and *fail-count*. The *operability* is the ratio of *no-count* and *f*.

$$success\text{-}rate = success\text{-}count / (success\text{-}count + fail\text{-}count)$$
$$operability = no\text{-}count / f$$

After we get the *success-rate* and *operability* of each pair-of-snapshots, we calculate the *success-rate* and *operability* of each candidate point. We first classify old snapshots by the candidate points they belong to. Then we sum *success-rate* or *operability* up. We treat the average of *success-rate* or *operability* as the *success-rate* or *operability* of the candidate point.

We don't need to calculate the *timeliness*, *success-rate* and *operability* of each candidate point in *new-code'*, because we only need to recommend the update point in the old version program.

When we get all these data, we weigh them when ranking candidate points and recommending update points.

4. AN EXPLORATORY STUDY

This section introduces the design of our exploratory study. The purpose of our study is to examine whether the above approach is feasible and effective in practice. We want to find out:

(1) Can we evaluate and recommend update points according to the properties that we defined?

(2) Whether there are correlations between the three properties?

(3) Whether the properties of the candidate points in some program structures are better than other candidate points?

4.1 Selection of Subjects

For the client-side software systems, the loss of time and data caused by terminating and updating are usually not serious. On the contrary, shutting down a server software system, such as an email server, may not only bring inconvenience to lots of customers and degrade user experience, but also lose a large amount of user data. So we choose to carry out our study on a server software system, Siena.

Siena (Scalable Internet Event Notification Architecture) [17] is an Internet-scale event notification middleware for distributed event-based applications deployed over wide-area networks. It is responsible for selecting notifications that are of interest to clients (as expressed in client subscriptions) and then delivering those notifications to the clients via access points. Siena is a popular open source Java software product, and one can get Siena and its test cases from SIR (Software-artifact Infrastructure Repository, http://sir.unl.edu/portal/index.php).

There are 26 classes and 138 fields in the 6 versions of program, 194 methods in Siena1.8, 185 methods in Siena1.9 and Siena1.10, 195 methods in Siena1.11, 196 methods in Siena1.14 and 192 methods in Siena1.15.

We obtained 8 versions of Siena (1.8, 1.9, 1.10, 1.11, 1.12, 1.13, 1.14 and 1.15) from SIR. We collected update information between the adjacent versions. There are five changed classes between 1.10 and 1.11, four changed classes between 1.8 and 1.9 also between 1.14 and 1.15, three changed classes between 1.13 and 1.14, two changed classes between 1.11 and 1.12, one changed class in 1.9 and 1.0 also between 1.12 and 1.13. We

Table 3. Source code and update information

Old-version/new-version	1.8/1.9	1.10/1.11	1.14/1.15
Class-number: updated/old/new	4/26/26	5/26/26	4/26/26
Field-number: updated/old/new	32/138/138	70/138/138	52/138/138
Method-number: updated/old/new	11/194/185	13/185/195	14/196/192
Code-lines: updated/old/new	612/1777/1756	646/1758/1794	752/1798/1784
Methods-with-point/unchanged-methods	167/183	166/182	166/182
Candidate-points/unchanged-codes	855/1165	820/1112	803/1046

conducted experiments with the three version update, which are 1.8 and 1.9, 1.10 and 1.11, 1.14 and 1.15, because the three version updates have the maximum three updated content.

Table 3 shows source code information and update information about the three version updates. The first row of Table 3 shows the old and new programs. The second row shows the number of updated classes and total number of classes in each version. The third row shows the number of updated fields and total number of fields in each version. We consider all the fields in an updated class as updated fields. The fourth row shows the number of updated methods (include constructors) and total number of methods in each version. The fifth row shows the updated lines of effective code and total lines of effective code in each version. When we count the lines of effective code, we only take the code in methods and constructors into consideration. So, the updated effective code is effective code in updated methods and constructors, the total lines of effective code are the total lines of effective code of all methods and constructors. The sixth row shows the number of unchanged methods (include constructors) with some candidate points and the total number of unchanged methods. The last row shows the number of candidate points and total lines of unchanged code (the effective cod in unchanged methods and constructors).

Table 4. Attributes of a snapshot

Property	Explanation
FileName	The name of Java file.
LineNo	The No. of this line of code.
MethodName	The name of method.
PointNo	The identifier of candidate point.
PointTimes	The times of passing this candidate point.
SnapshotCount	The count of snapshots captured for now.

4.2 Setting Candidate Points

In addition to collecting snapshots, we also need basic information about these snapshots. The attributes we recorded for each snapshot are shown in Table 4.

We matched old and new snapshots precisely by comparing these attributes. While *Catch.snapshot()* dumping snapshots, these attributes are recorded at the same time.

In our implementation, we tried to set candidate point behind every line ended with ";" and ignore some special lines. For example, in Table 2, we ignored the first line "void f() {" and the last line "}", which were also counted as effective code. Also, we didn't set candidate point behind "return;", "break;", "continue;" the conditions of branch statements ("if" or "switch"), or the conditions of loop statements ("for" or "while").
Table 3 demonstrates that, we selected 74.6% on average of unchanged lines of code and set candidate points, covering 91.2% on average of unchanged methods.

4.3 Running Test Cases

There are 581 test cases in the Siena project that we got from SIR. In order to ensure the validity of our experiment, we executed all the test cases in our experiments.
Our experiment configurations are as follows. The operating system is 64-bit Ubuntu Kylin 14.04 with 8GB RAM, Intel Core 3.40GHz 8-core CPU. We use OpenJDK 1.7 to perform our experiment.

Executing all these test cases on 6 programs, which took about one week, dumps a huge amount of snapshots showing in Table 5. TOS may fail a few times while analyzing pairs of snapshots due to bugs. Therefore, the count of snapshots that TOS analyzed is less than all of the snapshots.
TOS only can produce transformation functions for an updated field each time. We improve TOS to generate transformation functions for all fields each time, but the basic functionality has not changed.

4.4 Calculating Properties of Each Point

In section 3.1, we introduced three properties for candidate points, namely *timeliness*, *success-rate* and *operability*. We use algorithm 1 in Table 6 to calculate the *timeliness* and algorithm 2 in Table 7 to calculate *success-rate* and *operability*. To be sure, if there are no objects of any updated class in the snapshot, the *operability* of this snapshot is *1*. If the *operability* is *1*, we do not need to

Table 5. The number of snapshots

Update	1.8 to 1.9		1.10 to 1.11		1.14 to 1.15	
Version	1.8	1.9	1.10	1.11	1.14	1.15
Count of snapshots	572690	572831	653849	653802	663370	662047
Count of snapshots TOS analyzed	572566		653755		661961	

generate transformation functions, and we consider the *success-rate* as *1*.

Because of the huge amount of data, the whole progress of computing *timeliness*, *success-rate* and *operability* takes about 7 hours.

Table 6. Computing timeliness

Algorithm 1: compute *timeliness*
Input: collections-of-snapshots of old program
Output: the *timeliness* of each candidate points
1. Start;
2. Iterate through each collections-of-snapshots
3. Iterate through each snapshot;
4. Add *1* to the point which the snapshot belongs to;
5. Iterate through each candidate points;
6. Compute the ratio of snapshots number and total number;
7. End;

Table 7. Computing success-rate and operability

Algorithm 2: compute *success-rate* and *operability*
Input: candidate points, collections-of-snapshots
Output: *success-rate* and *operability* of each candidate points
1. Start;
2. Iterate through each collection-of-snapshots
3. Iterate through each pair-of-snapshots;
4. TOS generating transformer function for each updated fields;
5. Compute *success-rate* and *operability* of this snapshot;
6. Add *1* to the point which the snapshot belongs to;
7. Add the *success-rate* of this snapshot to the point;
8. Add the *operability* of this snapshot to the point;
9. Iterate through each candidate points;
10. Compute the average of *success-rate* of this point;
11. Compute the average of *operability* of this point;
12. End;

5. EXPERIMENT RESULTS

5.1 Feasibility of the Technique

We performed experiments on three real version updates of Siena. Before this section, we have introduced our experiment configurations. In this section, we will present the result of experiment and tries to answer question (1):

(1) Can we evaluate and recommend update points according to the properties that we defined?

When we tried to evaluate candidate points based on the result, we found that there existed some *null* points. The three properties (*timeliness*, *success-rate* and *operability*) of a *null* point were all *0*, which meant this point was never reached during executing test cases. We filtered out these *null* points. Then we found another problem. Although the three properties of left candidate points were not *0*, but some candidate points, called *bad* points, had extreme properties. For example, the *timeliness* of a bad point may be 0.99 (close to 1), but the *success-rate* of the bad point is very small (close to 0).

As we all know, there are few objects when a program starts. Therefore the *operability* and *success-rate* of candidate points in starting period is usually higher or even *1*. However, the program only starts once. So the *timeliness* of candidate points in starting period are usually small or even near-zero. If we rank candidate points by giving the highest priority to *operability*, some of the top-ranking points are actually *bad* choices.

Bad and *null* points would affect our results to a certain extent. Thus, we filtered out *bad* points too in the following steps.

After excluding *null* points, we counted the number of left points, calculated the sum of each property and computed the average of each property. We took the average as *threshold* and exclude points if one or more properties were smaller than the *threshold*. But this threshold would have filtered out most or ever all candidate points. So we took the half of average as the *threshold*. The *threshold* could be more accurate, and we will make further attempts in our future work.

Table 8 shows the number of null points and bad points in each update. After filtering out *null* and *bad* points, there are 91 candidate points left in update from 1.8 to 1.9, 94 candidate points left in update from 1.10 to 1.11 and 49 candidate points left in update from 1.14 to 1.15. We will recommend update points for each update.

Each snapshot has three properties. Each property can suit different needs. If we want to finish updating timelier, we can give more weight to *timeliness*. Similarly, by weighting *success-rate* more we can make the transformation function easier to develop, and thus be more confident about the correctness of dynamic update.

Table 8. Excluding candidate points

Update	1.8 to 1.9	1.10 to 1.11	1.14 to 1.15
Candidate points	855	820	803
***Null* points**	589	550	578
***Bad* points**	175	176	176
Remaining points	91	94	49

Table 9 shows the update points recommended by our approach. We take Siena1.8-1.9 as an example and consider timeliness first and *success-rate* second. From the result, the three properties can distinguish each update points. We can use the three properties to evaluate and recommend update points for programmers. After programmers getting our suggestions, they can determine the update points according to their judgments.

We check these points in each update according to our understandings of the logic and runtime state of each version and find out that dynamic updates can be applied at these update points and the executions after updating are as our expected.

5.2 Correlations between Properties

In this section, we tries to answer question (2):

(2) Whether there are correlations between the three properties?

As a matter of experience, the higher of *operability*, the less operations are needed to generate transformation functions. Therefore, we propose a hypothesis:

Table 9. Recommending 5 update points

PointNo	FileName	Class	MethodName	*Timeliness*	*Success-rate*	*Operability*
391	SENP.java	ByteBuf	bytes	0.1212	0.5947	0.678
385	SENP.java	ByteBuf	append	0.066	0.6457	0.6989
387	SENP.java	ByteBuf	append	0.0526	0.5394	0.648
398	SENP.java	Tokenizer	currByte	0.0472	0.6698	0.7152
399	SENP.java	Tokenizer	currByte	0.0472	0.6698	0.7152

H1: There is a positive correlation between *operability* and *success-rate*.

We calculate the *correlation* between each two properties. Because *null* points have great effect on *correlation*, we exclude *null* points before calculating *correlations*.

Table 10 shows the *correlations* we get. It indicates that generally there is neither correlation between *timeliness* and *success-rate*, nor correlation between *timeliness* and *operability*. As to the correlation between *success-rate* and *operability,* the result is significantly positive for updates from 1.8 to 1.9 and 1.10 to 1.11, and weakly positive (although very close to significant level) for the update from 1.10 to 1.11.

Table 10. Correlations between properties

Update	1.8 to 1.9	1.10 to 1.11	1.14 to 1.15
Timeliness vs ***Success-rate***	-0.0386	-0.0770	-0.0185
Timeliness vs ***Operability***	-0.1442	-0.1531	-0.1492
Success-rate vs ***Operability***	0.5916	0.6711	0.4951

In summary, hypothesis **H1** is confirmed by the result.

The positive correlation between *success-rate* and *operability* suggests that it may be unnecessary to measure both of them in recommending update points. The independence between *timeliness* and *success-rate*/*operability* suggests that it is possible to select update points that can make the dynamic update both timely and the transformation functions easy to write.

5.3 Correlation between Properties and Program Structure

In this section, we tries to answer question (3):

(3) Whether the properties of the candidate points in some program structures are better than other candidate points?

In the following part, "special statements" means branch statements ("if" and "switch") or loop statements ("for" and "while"), "common statements" means the others.

After we get the properties of every candidate points, we iterate through every unchanged method and find out whether there are special statements in the unchanged method. If there are special statements, we first calculate the average of every property of all candidate points in special statements, then calculate the average of every property of all candidate points in common statements. Afterwards, we compare the relationship between common statements and special statements, in the same methods.

Intuitively, *timeliness* of update points in loop statements may be higher than those in common statements; and those in branch statements may be lower than in common statements. Therefore, we propose two hypotheses:

H2: In same methods, the *timeliness* of candidate points in loop statements is higher than those in common statements.

H3: In same methods, the *timeliness* of candidate points in branch statements is lower than those in common statements.

In the update from 1.8 to 1.9, we find 18 unchanged methods have both common and special statements, in update from 1.10 to 1.11, we find 18 and in update from 1.14 to 1.15, we find 17. Totally, there are 43 unchanged methods have both common and special statements in our experiment.

Table 11 presents the comparisons of properties between common and special statements.

In the second row, we use abbreviations. "C" stands for "common statements" and "S" stands for "special statements". "C=S" in "***Timeliness***" column means that, *timeliness* of candidate points in common statements is equal to those in special statements. And "C>S" or "C<S" means the properties of candidate points in common statements are greater or smaller than those in special statements.

From Table 11, we can observe that, 37 unchanged methods have loop statements and 32 unchanged methods have branch statements. Because there can exist both loop and branch statements in a method, the sum of 32 and 37 is greater than 43 unchanged methods.

In "**Branch statements**" row, "***Timeliness***" column, there are 32 of "C>S", which means the *timeliness* of branch statements is smaller than common statements in all 32 unchanged methods

Table 11. Comparing special and common statements

Properties	*Timeliness*			*Success-rate*			*Operability*		
Comparisons	C=S	C>S	C<S	C=S	C >S	C<S	C=S	C >S	C<S
Loop statements	0	15	22	3	26	8	5	28	4
Branch statements	0	32	0	0	29	3	0	27	5

with branch statements. In 100% of unchanged method with branch statements, the *timeliness* in branch statements is smaller than those in common statements. Therefore, we accept hypothesis **H3**.

However, when we try to verify **H2**, we cannot give a definitive conclusion from Table 11. In "**Loop statements**" row, "***Timeliness***" column, we can see that 22 "C<S" are satisfy **H2**, but 15 "C>S" are not. We check these 15 methods and find that, the 15 unchanged methods have both loop statements and branch statements. And some of loop statements are in branch statements and the other loop statements have branch statements in them, that is the why *timeliness* in common statements are greater than loop statements. Therefore, we cannot accept **H2**, but we get another conclusion: In the same methods, the *timeliness* of candidate points in loop statements, which have no branch statements or are not in branch statements, is higher than those in common statements.

For *success-rate* and *operability*, we can conclude that, common statements have much higher values than special statements. This conforms to the intuition that states at theses points are transient and unstable, and thus the transform functions for them are difficult to develop.

This study clearly indicates that the program structures have some effect on properties of candidate points. In our future work, we plan to improve our approach by taking program structures into account.

6. DISCUSSION

6.1 Efficiency

In order to recommend update points with better update timeliness and less state transformation complexity, we considered as much candidate points as we could. In our experiment, we set candidate points in about 74% unchanged lines of valid code, which covered 91.2% of unchanged methods. In the following steps of the experiment, executing test cases, collecting snapshot at these candidate points and analyzing the properties of each candidate point were all time-consuming. It took about 8 days to get the results.

We spent little effort to optimize the execution performance of the experiment in this preliminary study, but we believe there is a large room for improvement.

Table 12. Equivalent update points

```
void main(){
    point_1;      f();
    point_2;      g();
    point_3;      h();
}
```

For example, we can borrow the idea of "equivalent update points" used in Hayden et al. [18]. In that work they proposed an efficient, systematic testing methodology for dynamically updateable software. To avoid duplicated testing they group semantically equivalent update points together and just test one of them. In our approach, we also just need to evaluate one update point in an equivalent group. For instance, in Table 12, we assume that f(), g() and h() do not call any other functions. And *point_1*, *point_2*, *point_3* and *point_4* are update points. In an update, f() and g() remain the same, h() is changed. Whether we apply the update at *point_1*, *point_2* or *point_3*, the behavior of this program is the same. The main() calls f() and g(). And these calls will point to the old version. On the other hand, the calls to h() will point to the new version. So *point_1*, *point_2* and *point_3* are equivalent points.

Initial estimation with our subject shows that the method can reduce equivalent update tests and equivalent update points for about 90%. We plan to adopt this optimization in our future work.

6.2 Test Case And Real-World Execution

Our approach and experiment executing test cases to collect snapshots and then analyze properties of candidate points. The most practical way is getting a large amount of different user's operation trace and then collecting snapshots from these trace. However, it is extremely difficult to obtain these data. After the comprehensive balance, we decide to use a set of test cases, which can execute on both old and new programs. This will lead to little deviation from real-world execution. But we think it is acceptable. If one wants to use our approach to get more precise results, the real-world execution can be taken into consideration.

7. CONCLUSION AND FUTURE WORK

In this paper, we conceived an approach to automatic recommendation of update points for dynamic software updating. Some preliminary experiments were carried out to examine the feasibility and effectiveness of our approach and explore some relevant issues.

We are working on further improvements to this approach such as a better strategy to select candidate update points, performance optimizations and more experiments with real world subjects to evaluate the correctness and effectiveness of our approach.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful suggestions and comments. This work was supported in part by National Basic Research 973 Program (Grant No. 2015CB352202), and National Natural Science Foundation (Grant Nos. 61472177, 91318301, 61321491, 61361120097) of China.

REFERENCES

[1] Subramanian, S., Hicks, M., & McKinley, K. S. (2009). Dynamic software updates: a VM-centric approach (Vol. 44, No. 6, pp. 1-12). ACM.

[2] Hayden, C. M., Smith, E. K., Hardisty, E. A., Hicks, M., & Foster, J. S. (2012). Evaluating dynamic software update safety using systematic testing. Software Engineering, IEEE Transactions on, 38(6), 1340-1354.

[3] Hicks, M., Moore, J. T., & Nettles, S. (2001). Dynamic software updating (Vol. 36, No. 5, pp. 13-23). ACM.

[4] Magill, S., Hicks, M., Subramanian, S., & McKinley, K. S. (2012, October). Automating object transformations for dynamic software updating. In ACM SIGPLAN Notices (Vol. 47, No. 10, pp. 265-280). ACM.

[5] Neamtiu, I., & Hicks, M. (2009, June). Safe and timely updates to multi-threaded programs. In ACM Sigplan Notices (Vol. 44, No. 6, pp. 13-24). ACM.

[6] Gupta, D., Jalote, P., & Barua, G. (1996). A formal framework for on-line software version change. Software Engineering, IEEE Transactions on, 22(2), 120-131.

[7] Neamtiu, I., Hicks, M., Foster, J. S., & Pratikakis, P. (2008, January). Contextual effects for version-consistent dynamic software updating and safe concurrent programming. In ACM SIGPLAN Notices (Vol. 43, No. 1, pp. 37-49). ACM.

[8] Arnold, J., & Kaashoek, M. F. (2009, April). Ksplice: Automatic rebootless kernel updates. In Proceedings of the 4th ACM European conference on Computer systems (pp. 187-198). ACM.

[9] Mitchell, N., & Sevitsky, G. (2003). LeakBot: An automated and lightweight tool for diagnosing memory leaks in large Java applications. In ECOOP 2003–Object-Oriented Programming (pp. 351-377). Springer Berlin Heidelberg.

[10] Bierman, G., Parkinson, M., & Noble, J. (2008). UpgradeJ: Incremental typechecking for class upgrades. In ECOOP 2008–Object-Oriented Programming (pp. 235-259). Springer Berlin Heidelberg.

[11] Chen, H., Yu, J., Chen, R., Zang, B., & Yew, P. C. (2007, May). Polus: A powerful live updating system. In Proceedings of the 29th international conference on Software Engineering (pp. 271-281). IEEE Computer Society.

[12] Altekar, G., Bagrak, I., Burstein, P., & Schultz, A. (2005, August). OPUS: Online Patches and Updates for Security. In Usenix Security (Vol. 5).

[13] Malabarba, S., Pandey, R., Gragg, J., Barr, E., & Barnes, J. F. (2000). Runtime support for type-safe dynamic Java classes (pp. 337-361). Springer Berlin Heidelberg.

[14] Gu T., Cao C., Xu C., et al. Javelus: A Low Disruptive Approach to Dynamic Software Updates[C]//Software Engineering Conference (APSEC), 2012 19th Asia-Pacific. IEEE, 2012, 1: 527-536.

[15] Matozoid, Java 1.7 parser and Abstract Syntax Tree. https://github.com/matozoid/javaparser, 2007

[16] ORACLE, Java Documentation, Java Platform, Standard Edition Tools Reference. http://docs.oracle.com/javase/8/docs/technotes/tools/unix/, 1993.

[17] Software-artifact Infrastructure Repository, Object Biography: Siena, http://sir.unl.edu/portal/bios/siena.php, Jul. 20, 2014

[18] Hayden, C. M., Hardisty, E. A., Hicks, M., & Foster, J. S. (2009, October). Efficient systematic testing for dynamically updatable software. In Proceedings of the 2nd International Workshop on Hot Topics in Software Upgrades (p. 9). ACM.