Automated Recommendation of Dynamic Software Update Points

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

ABSTRACT

Due to the demand for bugs fixing and features enhancements, developers inevitably need to update in-use software systems. Instead of shutting down a running software 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 *state transformer* 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 state transformer functions; and to strike a good balance between them requires a deep understanding of both versions of the program. In this paper, we propose a technique that automates the recommendation of update points. We carried out a preliminary study about the feasibility and effectiveness of this technique. By conducting a set of experiments based on an actual DSU case, we found that our technique is automatic, wide-covered and efficient.

Categories and Subject Descriptors

B.5.2 [Design Aids]: Automatic synthesis; I.2.5 [Programming Languages and Software] Expert system tools and techniques;

General Terms

Design, Languages, Experimentation.

Keywords

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

1. INTRODUCTION

Software can help us finish various tasks. However, because of some reasons, there always are vulnerabilities in software. Therefore, programmers must update software constantly.

General software update schemas need software to be static. Terminating the software first, if the software is running. Modifying the changed parts and then restarting the software. However, in some situations, the stop-and-restart update schema is not acceptable.

Conference'04, Month 1–2, 2004, Nanjing, Jiangsu, China.

DSU is a generic technique, which can alleviate these problems by patching programs on the fly. In general, DSU systems need to determine update points, generate state transformers and prepare the dynamic patch. Update points decide the timing for doing update. State transformer functions migrate the old program state at the update points to the new program. Dynamic patch [1,8,9,14] indicates the update information. Most DSU systems (e.g. Jvolve [1] and Javelus [14]) can produce default transformer functions. These default transformers assign default values to new fields (e.g. **0** for **int** and **null** for **String**).

Some existing DSU systems (i.e. Jvolve[1] and Javelus[14]) apply update at DSU *safe points* [1], which are a subset of VM safe points. JVM can perform thread scheduling and garbage collection at a VM safe point. Jvolve detects the stacks at a VM safe point to find if there are *restricted* methods. If there are not, Jvolve takes this VM safe point as a DSU *safe point* and applies the update. If there are *restricted* methods, Jvolve would defer the update. Restricted methods include updated methods, methods which refer to updated classes and methods which are blacklisted by users.

But imagine that, there exists an email server program which has a *Listen* method. The *Listen* method should always running to listen users' access. But in an update, we want to use Jvolve to modify the *Listen* method. Since the *Listen* is always running, Jvolve can never reach a DSU safe point. Jvolve also can determine DSU *safe points* manually, but it is time consuming and error-prone.

In this paper, we present an automated technique that can recommend update points and perform experiments to evaluate it.

Above all, we define some properties for candidate points. Our approach evaluates and recommends update points according to these properties.

Definition 1. *Immediacy* means the frequency of passing by a candidate point during the execution of program.

Definition 2. *Success-rate* means the proportion of generating transformers successfully while generating for all updated fields at a candidate point.

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

While program is running and an update is available, we usually want to apply the update quickly. We need update points which can be passed by frequently. Update points with higher *immediacy* satisfy this demand. When we apply the update, we need to transformer old objects to new version program. Assume there are

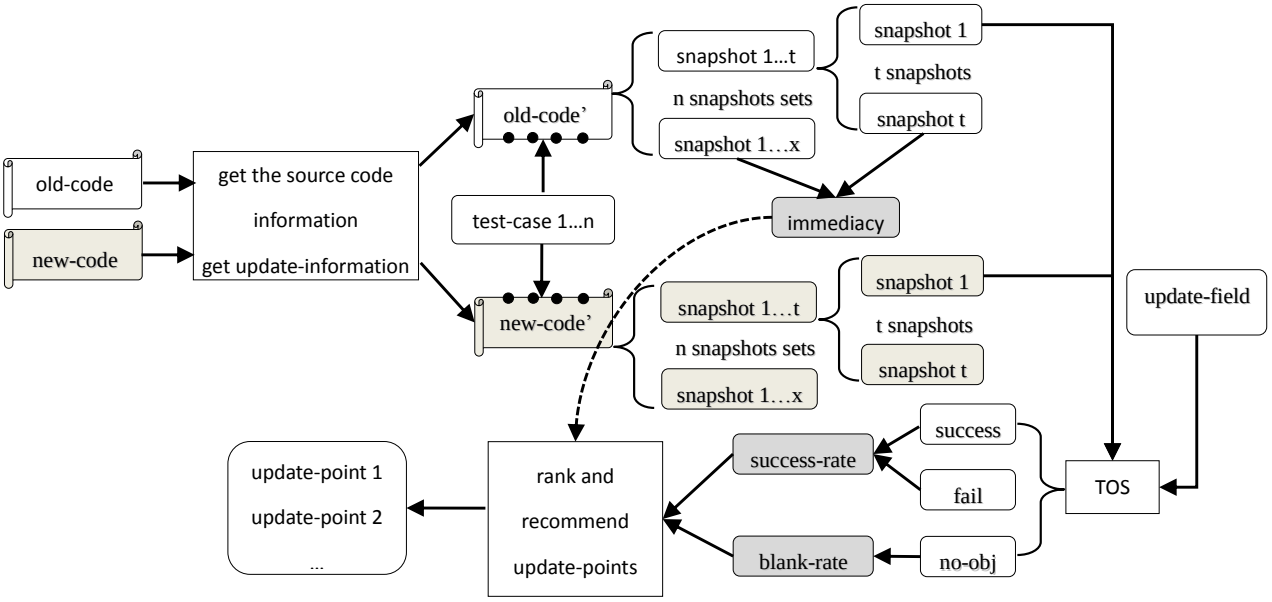

Figure 1. Automated recommendation of update points

c updated classes and *f* updated fields, which means we need *f* transformer functions to migrate all objects belong to *c* updated classes. In our approach, we need an automated tool (e.g. Targeted Object Synthesis [4]) to generate transformers for updated fields. If the number of updated classes with objects being created is little at a candidate point, we just need to generate transformers for few updated fields and to migrate a small number of objects to new version program at this candidate point. Moreover, automated tool (e.g. Targeted Object Synthesis) may fail sometimes when producing transformers. So the larger proportion of success, the easier of producing transformers. Therefore, applying update at update points with high *success-rate* and *blank-rate* is more likely to succeed.

We want to recommend update points with nice properties. In our experiment, we set three *thresholds* to filter out inappropriate candidate points. Because our experiment is preliminary, the *thresholds* we use is very simple. We use half of the average of each properties as *thresholds*. Judging from the results, the simple *thresholds* works very well. But we will make *thresholds* more precise in the future work.

Our automated approach works in the following steps: analyzing the source code of old and new programs to get class information and update information; inserting candidate points in the unchanged methods in both versions; running the same test cases on old and new programs and catching memory snapshots while passing by a candidate point; calling Targeted Object Synthesis (TOS) [4] to generate transformers for each updated fields and collecting all the results; computing the properties (*immediacy*, *success-rate*, *blank-rate*) of each candidate point and recommending update points by analyzing these properties.

The main contributions of this paper are:

1) An automated technique for recommending update points, and
2) An implementation of this automated technique, and
3) A preliminary evaluation of this automated technique with a case study.

The rest of this paper is organized as follows. Section 2 introduces DSU and TOS briefly. In Section 3, we describe our technique specifically. Section 4 presents the implementation of our technique and experiments setup. We show our experiment results in Section 5. And section 6 shows the related work. Finally, Section 7 concludes this paper and introduces the future work.

2. TARGETED OBJECT SYNTHESIS

2.1 Dynamic software updating

Most of DSU systems can change updated code automatically during the runtime of old version program. In order to running the new version program normally, DSU systems also need to migrate the state of old version program to new version program and make it compatible with new version program. Transformer functions can do this work. Some DSU systems, like Jvolve [1] and Javelus [14] can produce default transformer functions automatically, by analyzing the bytecode of old and new programs. However, to make them easy to use, Jvolve and Javelus minimize the effort to generate transformers. Therefore, these default transformers have imperfection: assigning default value to new fields (e.g. **0** for **int** and **null** for **String**).

In practice, these changed fields should hold a specific value instead of default value. For example, if we change the definition of a field named *EmailAddress*, which should save the email address, in an email server software. The default transformers assign **null** for *EmailAddress*. But we all know that *EmailAddress* in 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 transformer sometimes.

It is easy to produce default transformers and rarely fail. In order to update program more meaningfully, we should employ more practical transformers. Moreover, our approach is an automated

recommendation of update points, so we want to use an automated tool to generate transformers.

2.2 Targeted Object Synthesis

Targeted Object Synthesis (TOS) [4] can automatically produce transformers for updated fields. TOS extracts old and new objects of updated classes from old and new memory snapshots separately, then analyzes objects to produce transformers. The progress of TOS can be 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 and synthesizes transformers for them. These transformers can assign more valuable and practical values to updated field as we mentioned in **2.1**.

Each time TOS generate transformers for an updated field, there would be three kind of results: generate successfully, unsuccessfully or there is no objects of updated class. We consider the first result as *success*, the second as *fail* and the third as *no-obj*.

TOS only can produce transformers for an updated field each time. We improve TOS to generate transformer functions for all fields, each time being called, but the basic functionality has not changed.

3. AUTOMATED RECOMMENDATION OF UPDATE POINTS

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

Update points can be selected manually. For example, Jvolve can apply update at DSU safe points, which can be determined automatically or manually. Unfortunately, the cases with manual intervention are usually challenging to be right. Furthermore, the automation mechanism cannot be sure to reach a DSU safe points.

We present an automated recommendation of update points, shown in Figure 1, to address these problems. Our approach can evaluate candidate points and recommend some update points, given source code of the old and new versions and some test cases. By now, we implement our approach for Java program, but we believe that our approach can be applied to other programming languages and embedded in most of DSU systems. 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 the source code's information, which contains classes, fields, methods and so on. 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 passing a candidate point. The memory snapshot contains all live objects which are created during the execution of Java programs.

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;
	}

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 write *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 the exactly 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 *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 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 one test case on both versions of 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 *immediacy* 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 *immediacy* of candidate point

is the ratio of *candidate-count* and *total-count*. The higher of *immediacy*, the timelier of starting applying the update.

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

After collecting all the snapshots of both *old-code'* and *new-code'* and pairing them up, we will invoke TOS [4]. TOS automatically produce transformers 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 transformers for *f updated-fields*. We consider that producing transformers successfully as *success*, unsuccessfully as *fail*. Moreover, there may be no objects of the updated class in one pair-of-snapshots, and we consider this result as *no-obj*. After we finish generating transformers for updated fields, we count the number of each result, *success-count* for *success*, *fail-count* for *fail* and *no-count* for *no-object*. And 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 *blank-rate* is the ratio of *no-count* and *f*.

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

After we get the *success-rate* and *blank-rate* of each pair-of-snapshots, we calculate the *success-rate* and *blank-rate* of each candidate points. We first classify old snapshots by the candidate points they belong to. Then we sum *success-rate* or *blank-rate* up. We treat the average of *success-rate* or *blank-rate* as the *success-rate* or *blank-rate* of candidate point. The higher of *success-rate*, the more likely of finishing DSU successfully. The higher of *blank-rate*, the easier of performing DSU.

We don't need to calculate the *immediacy*, *success-rate* and *blank-rate* 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 evaluating candidate points and recommending update points.

4. EXPERIMENTS

This section introduces the design of our experiment. The purpose of our study is to verify that our technique proposed in this paper is effective and we also want to answer the following questions.

(1) When we recommend update points, how should we weigh the three properties?

(2) Does the automated technique, proposed in this paper really works and is applicable generally?

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

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

4.1 Selection of subjects

For the client software, the loss of time and data caused by terminating program's execution and updating them are usually not serious. On the contrary, shutting down server software, such as email server, will not only bring inconvenience to lots of customers, degrades the user experience, but also loses a large amount of user data. So we choose to carry out our experiment on server software, 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, 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 the more popular open source Java software, we get Siena and the test cases from SIR (Software-artifact Infrastructure Repository, http://sir.unl.edu/portal/index.php).

There are 8 versions of Siena (1.8, 1.9, 1.10, 1.11, 1.12, 1.13, 1.14 and 1.15) in the file we obtain from SIR. We collect 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, 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, 1.12 and 1.13. We conduct experiments with three version update: 1.8 and 1.9, 1.10 and 1.11, 1.14 and 1.15,

Table 3. Source code and update information

Old-version/new-version	**1.8/1.9**	**1.10/1.11**	**1.14/1.15**
Updated-classes/total-classes	4/26	5/26	4/26
Updated-fields/total-fields	32/138	70/138	52/138
Updated-methods/total-methods	11/194	3/185	14/196
Updated-codes/total-codes	612/1777	646/1758	752/1798
Methods-with-point/unchanged-methods	167/183	166/182	166/182
Candidate-points/unchanged-codes	855/1165	820/1112	803/1046

Table 4. 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	

because the three version updates have the maximum 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 all classes in each update. The third row shows the number of updated fields and total number of fields. We consider all the fields in an updated class as updated fields. The fourth row shows the number of updated methods (including constructors) and all methods (including constructors). The fifth row shows the updated lines of effective code and total lines of effective code. When we count the lines of effective code, we only take the code in methods and constructors into consideration, because we only can set candidate points in 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 (include constructors). The last row shows the number of candidate points and total lines of unchanged effective code.

Table 5. 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 collect snapshots, we also need basic information about these snapshots. The attributes we record for each snapshot are shown in Table 5.

We match 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 try to set candidate point behind every line ended with ";" and ignore some special lines. For example, in Table 2, we ignore the first line "void f() {" and the last line "}", which are also counted as effective code. Also, we do not set candidate point behind "return;", "break;", "continue;" and the conditions of branch statements ("if" and "switch") or loop statements ("for" and "while").
Table 3 demonstrate that, we select 74.6% unchanged lines of code on average and set candidate points behind these lines, covering 91.2% unchanged methods on average.

4.3 Running test cases

There are 581 test cases in the Siena project that we get from SIR. In order to ensure the validity of our experiment, we execute all the test cases in our experiments, which takes about one week and dumps a huge amount of snapshots showing in Table 4. TOS may fail a few times due to bugs. Therefore, the count of snapshots that TOS analyzed is less than all snapshots.
If we save all these snapshots, there will be no enough space. So when we get two corresponding collections of snapshots, we invoke TOS to analyzing the pairs of snapshots in the two collections. Afterwards, deleting the two collections, and then execute the next test case.

Table 6. Computing immediacy

Algorithm 1: compute *immediacy*
Input: collections-of-snapshots of old program
Output: the *immediacy* 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;

4.4 Calculating properties of each point

In section 2, we introduced three properties for candidate points, namely *immediacy*, *success-rate* and *blank-rate*. And we think that, the better of *immediacy*, the timelier of DSU operation; the better of *success-rate*, the easier by generating transformer functions and finishing DSU; the better of *blank-rate*, the less of generating transformer functions and the less of transforming old objects.

Table 7. Computing success-rate and blank-rate

Algorithm 2: compute *success-rate* and *blank-rate*
Input: candidate points, collections-of-snapshots
Output: *success-rate* and *blank-rate* 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 *blank-rate* 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 *blank-rate* 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 *blank-rate* of this point;
12. End;

We use algorithm 1 in Table 6 to calculate the *immediacy* and algorithm 2 in Table 7 to calculate *success-rate* and *blank-rate*. To be sure, if there are no objects of any updated class in a snapshot, the *blank-rate* of this snapshot is *1*. If the *blank-rate* is *1*, we do not need to generate transformer functions, and we consider the *success-rate* as *1*.

Because of the huge amount of data, the whole progress of computing three properties takes about 7 hours.

4.5 Experiment environment

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 perform our experiment on OpenJDK 1.7.

5. EXPERIMENT RESULTS

5.1 Effectiveness of this technique

We perform experiments on three update to Siena. Before this section, we have introduced our experiment conpletely. In this section, we will present the result of experiment and announce the answer of question (1) and (2):

(1) When we recommend update points, how should we weigh the three properties?

(2) Does the automated technique, proposed in this paper really works and is applicable generally?

When we try to evaluate candidate points based on the result, we find that there exists some *null* points. The three properties (*immediacy*, *success-rate* and *blank-rate*) of a *null* point are all *0*, which means this point is never reached during executing test cases. We filter out these *null* points. Then we find another problem. Some candidate points, called *bad* points, have disproportionate properties.

We all know that, there are few objects while program initiating. Therefor the *blank-rate* and *success-rate* of candidate points in initiating period is usually higher or even *1*. However, the program only initiates once. So the *immediacy* of candidate points in initiating period is usually small or even near-zero. If we rank candidate points by giving first priority to *blank-rate*, some of the top-ranking points are *bad* points.

Apart from the above reasons, there are still some *bad* points. These points would affect our results to a certain extent. Thus, we filter out *bad* points too by following steps.

After excluding *null* points, we count the number of left points, calculate the sum of each property and compute the average of each property. We want to take the average of three properties as *thresholds* and exclude points if one or more properties are smaller than *thresholds*. But this *thresholds* would filter out most or ever all candidate points. So we take the half of average as the *threshold*. The *threshold* could be more accurate, and we will make further attempts in our future work.

In Table 8, we show the number of null points and bad points in each update. After we filter out *null* and *bad* points, there are 91

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

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 evaluate these candidate points and recommend update points for each update.

Each candidate point has three properties. Each property can suit different needs. If we want to start updating timelier, we can prioritize *immediacy* first. Also, we can take *success-rate* into consideration first, in order to update programs more valid.

In our experiment, we rank candidate points based on three properties. Then we choose the top points as update points.

The update points, recommended by our approach, are safe and efficient. Because the update points are located at unchanged methods, it is safe to perform dynamic updating while unchanged methods are executing on stack [1, 14]. Meanwhile, we define three properties and we can select timelier and more effective update points with these properties.

Moreover, our approach are adapted to other Java programs. With old and new programs, a suite of test cases and a tool like TOS, our approach can analyze the source code and recommend update points automatically, and these update points are safe, efficient and meet different requirements.

5.2 Correlation between properties

In this section, we want to find answers to question (3):

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

As a matter of experience, the higher of *blank-rate*, the less operations are needed to generate transformer functions and the *success-rate* may be higher. Therefore, we propose a hypothesis:

H1: It is a positive correlation between *blank-rate* and *success-rate*.

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

Table 9 shows the *correlations* we get. And we can find that: it is no correlation between *immediacy* and *success-rate*, also no correlation between *immediacy* and *blank-rate*. In updates from 1.8 to 1.9 and 1.10 to 1.11, it is a significant positive correlation between *success-rate* and *blank-rate*, and in update from 1.10 to 1.11, it is a weak positive correlation (very close to significant correlation) between *success-rate* and *blank-rate*.

In summary, hypothesis **H1** is verified. Therefore the answer to questions (3) is:

Table 9. Correlations between properties

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

It is a significant positive correlation between *success-rate* and *blank-rate*. And no correlation between *immediacy* and *success-rate*, *immediacy* and *blank-rate*.

In our future work, we want to formulate a sound strategy to evaluate candidate points. Since we noticed the correlation between *blank-rate* and *success-rate*, we will weigh the correlation between them.

5.3 Correlation between properties and program structure

In this section, we will answer the question (4):

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

In the following part, "special statements" means branch statements ("if" and "switch) and loop statements ("for" and "while"), "common statements" means the others. We only take branch statements and loop statements into consideration for now.

After we get the properties of every candidate points, we iterate through every unchanged method and find out that whether there is 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.

As a matter of experience, in loop statements, the *immediacy* may be higher than common statements; in branch statements, the *immediacy* may be lower than common statements. Therefore, we propose two hypotheses:

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

H3: In the same methods, the *immediacy* 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 10 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 "***Immediacy***" column means that, *immediacy* 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 10, we can tell 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, "***Immediacy***" column, there are 32 of "C>S", which means the *immediacy* of branch statements is smaller than common statements in all 32 unchanged methods with branch statements. In 100% of unchanged method with branch statements, the immediacy in branch statements is smaller than those in common statements. Therefore, we verify hypothesis **H3**.

However, when we try to verify **H2**, we cannot give a definitive conclusion from Table 10. In "**Loop statements**" row, "***Immediacy***" 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 *immediacy* in common statements are greater than loop statements. Therefore, we cannot verify **H2**, but we get another conclusion: In the same methods, the immediacy 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 blank-rate, we can conclude that, common statements have much better data than special statements.

Since we notice the program structures have some effect on properties of candidate points, we will make some improvement in our approach in future work.

6. RELATED WORK

Hayden et al. [18] proposed an efficient systematic testing methodology for dynamically updateable software. They change the standard system test cases into update tests, which execute as before. Also, their approach can find equivalent update points during lots of update points.

In Table 11, we assume that f(), g() and h()do not call any other functions. And *point_1*, *point_2* and *point_3* 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.

The methodology can reduce equivalent update tests and equivalent update points for 90% on average.

In our future work, we will try this methodology on our experiments to improve efficiency.

7. CONCLUSION and FUTURE WORK

In this paper, we propose a technique to automatically recommend some update points for dynamic software updating. We design some experiments to validate the effectiveness of our technique and explore some relevant issues. As far as we know, there is no similar work to define the properties of update points and recommend update points based on these properties.

We are working on further improvements of this approach such as a sound strategy to evaluate candidate points and recommend update points and more experiments with mores subjects to

Table 10. Comparing special and common statements

Properties	***Immediacy***			***Success-rate***			***Blank-rate***		
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

evaluate the correctness and effectiveness of our approach.

Table 11. Equivalent update points

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

ACKNOWLEDGMENTS

This work was supported in part by National Basic Research 973 Program (Grant No. 2015CB352202), and National Natural Science Foundation (Grant Nos. 61472174, 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, Javaparser. https://github.com/matozoid/javaparser, Aug. 30, 2014

[16] ORACLE, Java Documentation. http://docs.oracle.com/javase/8/docs/technotes/tools/unix/, Aug. 30, 2014

[17] SIR, 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.