KupC: A Formal Tool for Modeling and Verifying Dynamic Updating of C Programs

Jiaqi Qian[1], Min Zhang[1(✉)], Yi Wang[2], and Kazuhiro Ogata[3]

[1] Shanghai Key Lab of Trustworthy Computing, ECNU, Shanghai, China
zhangmin@sei.ecnu.edu.cn
[2] GCCIS, Rochester Institute of Technology, Rochester, NY, USA
[3] Japan Advanced Institute of Science and Technology, Nomi, Japan

Abstract. Dynamic Software Updating (DSU) is a useful technique for updating running software without incurring any downtime. Its correctness must be guaranteed because updating a running software is a complicated and safety-critical process. In this paper, we present a formal tool called KupC for modeling and verifying dynamic updating of C programs. The tool is built on $\mathbb{K}$–a formal semantic framework for programming languages. We formalize a patch-based dynamic updating mechanism in $\mathbb{K}$ based on the formal executable operational semantics of C. The formalization automatically yields an interpreter and several verification tools, which can be used to formally analyze the correctness of dynamic updating for C programs. To our knowledge, KupC is the first formal tool for code-level verification of dynamic software updating.

1 Introduction

Software systems require frequent updating to fixate defects, improve performance, and add new features. For those systems providing 24×7 service commitment, Dynamic Software Updating (DSU) is a useful technique as it does not incur system downtime while updating [5]. Such systems are becoming prevalent with the diffusion of Internet of Things (IoT) and Cyber-Physical Systems (CPS), where additions, modifications, and removal of behaviors could be done in a quick and localized fashion. There is a comprehensive survey on DSU [10].

The difficulty of guaranteeing the correctness of dynamic updating is a fundamental barrier when we adopt this technique widely as expected. Correctness is crucial to those systems that need dynamic updating because they are usually safety-critical and highly-dependable. Meanwhile, dynamically updating a running software system is a complicated process, and it is difficult to predict

This work was supported by NSFC Project grants 61502171 and 61872146, and China HGJ Project under Grant 2017ZX01038102-002.

© The Author(s) 2019
R. Hähnle and W. van der Aalst (Eds.): FASE 2019, LNCS 11424, pp. 299–305, 2019.
https://doi.org/10.1007/978-3-030-16722-6_17

all possible updating results. In order to update a program successfully while it is running in practice one has to know everything about that program [6]. However, it still lacks effective methodologies and tools to help understand all possible behaviors of running programs caused by updating.

Formal methods are rigorous approaches to program verification. Some attempts have been made on applying formal methods to DSU [3,4]. The existing approaches suffer one or more difficulties as follows. In some approaches formalizing a dynamic update may require abstraction of target programs. Such abstraction is usually done manually. It requires both formal methods expertise and human intellection to interpret target programs. Some approaches [1,11] lack tool support while developing such tools needs substantial efforts.

To mitigate the above difficulties, we present a formal tool called KupC for modeling and verifying dynamic updating of C programs in this paper. KupC is built upon the formalization of a DSU tool called `Ginseng` [8] for C programs. We formalize the updating strategy of `Ginseng` atop the operational semantics of C in the formal semantic framework called $\mathbb{K}$ [9]. From the formalization, $\mathbb{K}$ automatically generates several tools that can be used for formal analysis of dynamic updating of C programs. According to our knowledge, KupC is the first tool for the code-level formal verification of dynamic software updating.

KupC has the following three features. (1) KupC is focused on the code-level verification of dynamic updating. It does not require any abstraction or transformation of target C programs that are subject to dynamic updating. (2) The verification functionalities of KupC are automatically generated from the formalization of dynamic updating mechanisms. No extra effort is needed on the implementation. (3) The formalization is built upon the operational semantics of the C language. One can easily develop similar tools for the formal analysis of dynamic updating of other languages such as Java and Python, whose operational semantics have already been formally defined in $\mathbb{K}$.

2 KupC Design

Patch-based DSU. Many DSU tools achieve dynamic updating by injecting patches into running programs [10]. A patch contains all updating contents, e.g., new functions and data. Figure 1 (left) is an overview of the patch-based updating process. An old-version program is first made updatable by attaching additional version information, wrapping user-defined types, and inserting possible updating points. They are achieved by the two operations called *Dependants Updating* and *Restriction Generating*. Next, a patch file *p1.c* is generated and complied by comparing the differences between old and new programs. After an update request is invoked, a DSU tool checks whether it is safe to inject the compiled patch whenever the running program reaches a pre-specified updating point. Safety means that the behavior of the updated program is consistent with the expectation. It is guaranteed by the adopted updating policies in DSU tools.

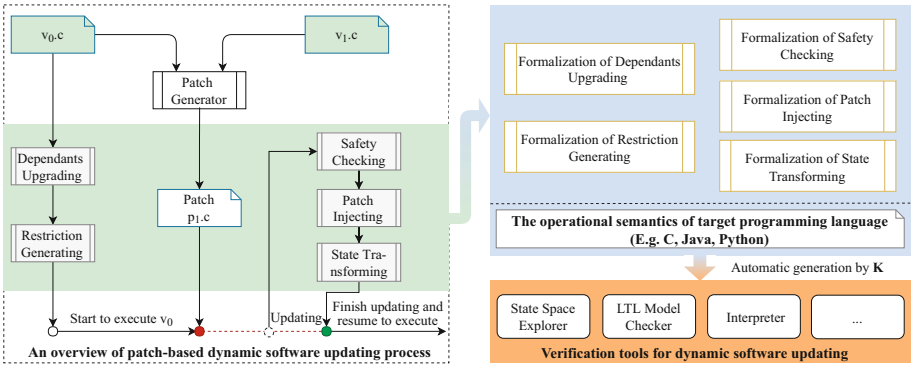


Fig. 1. Patch-based dynamic updating and its formalization using $\mathbb{K}$

If it is safe, the patch is injected and the running program state is transformed into the new version by a transformation function that is predefined in the patch. The patched program continues to execute from the new state. If updating at this point is not safe, the program continues to execute the old version.

It is worth mentioning that the entire updating process is atomically performed, that is, the execution keeps being suspended until the completion of the updating. Updating in an atomic manner is the most consistent approach that simplifies the updating process and reduces unexpected errors.

The $\mathbb{K}$ Framework. $\mathbb{K}$ [9] is a state-of-art semantic framework for programming languages. Many mainstream languages such as C and Java have been completely defined in $\mathbb{K}$. One only needs to focus on the formalization of an updating mechanism using the pre-defined operational semantics of the targeted language. After formalizing the updating mechanism, $\mathbb{K}$ automatically generates several analysis tools such as program interpreter, state space explorer, and model checker.

Formalization of dynamic updating strategy in $\mathbb{K}$. The basic idea of formalizing a dynamic updating mechanism using $\mathbb{K}$ is to formalize the functionalities of the mechanism on the basis of the operational semantics of the target programming language that the mechanism supports. The right part of Fig. 1 shows the formalization of the patch-based dynamic updating mechanism, consisting of the formalization of the five functionalities, respectively.

The functionalities of an updating mechanism are formalized by a set of rewrite rules. For instance, below is a rewrite rule that formalizes the function of checking the safety of updating a set of functions at an updating point *Loc*.

$$\left\langle TypeSafety(Loc, (\frac{F}{\cdot})\_) \cdots \right\rangle_k \quad \begin{array}{l} \langle \cdots Loc \mapsto (\_, \_, Re) \cdots \rangle_{restriction} \\ \langle \cdots F \mapsto T \quad F_{New} \mapsto T' \cdots \rangle_{types} \end{array}$$
$$when\ ((F \in Re) \wedge (T == T')) \vee (F \notin Re) \qquad (\textsc{SafetyChecking})$$

```
1 struct Road{
2   int dist;
3 };
4 struct City{
5   ... // node structure
6 };
7 struct Graph{
8   ... // Road+City..
9 };
10 struct Graph G;
11 void Shortest(int x){
12   ... // shortest path..
13 }
14 void PrintR(int x){
15   ... // print results..
16 }
17 void LoadG(){
18   ... // load graph data..
19 }
```

```
20 void Calculate(int x){
21   LoadG();
22   Shortest(x);
23 }
24 void Query(int x,int y){
25   /* point1 */
26   Calculate(x);
27   /* point2 */
28   PrintR(x,y);
29   /* point3 */
30 }
31 int main(){
32   ...
33   Query(0,6);
34   ...
35   Query(0,6);
36   ...
37 }
```

```
1 struct Road{ // modified structure
2   int dist;
3   int cost;  // new element
4 };
5 void Cheapest(int x){ // newly added
6   ... // new function
7 }
8 void PrintR(int x){ // modified
9   ... // print results and
10  ... // the cheapest path
11 }
12 void LoadG(){ // modified
13   ... // load new data
14 }
15 void Calculate(int x){  // modified
16   LoadG();
17   Shortest(x);
18   Cheapest(x);
19 }
```

Fig. 2. The snippets of old-version and new-version programs of a GPS application

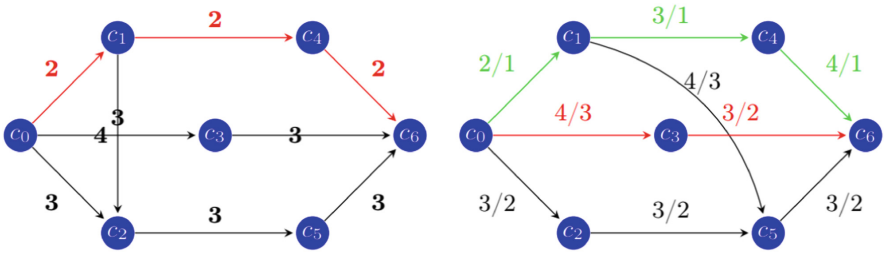
In the rule, a pair of brackets is a labeled *cell*, representing a piece of program execution information. $\frac{F}{\cdot}$ means F is deleted from the set if the conduction that follows the keyword *when* is true. The condition says that either F is updatable (represented by $F \notin Re$) or it is un-updatable at the point *Loc* but its types T and T' (before and after updating, respectively) are the same. Here, Re is the set of un-updatable contents at *Loc*. If the second argument of *TypeSafety* becomes an empty set, it means all the functions in the set are safe to update.

We totally defined 371 rewrite rules to formalize the updating mechanism of `Ginseng`. We tested the correctness of the rules using the example dynamic updating programs provided in `Ginseng`. These rules are seamlessly compiled by $\mathbb{K}$ together with the rules defined for the operational semantics of C [2]. The compilation yields the formal tool KupC which supports formal analysis of dynamic updating of C programs in various ways such as simulation, state exploration, and LTL model checking.

3 KupC Usage

KupC is equipped with an interpreter to *execute* updatable C programs, a state space explorer to search for all possible updating results, and an LTL model checker to verify temporal properties of dynamic updating. We demonstrate the usage of KupC using a dynamic updating to a GPS application. The tool, examples and a demo video are available https://github.com/dexter-qjq/KupC.

The program in Fig. 2 (left) is the old version of a GPS system. It calculates the shortest path. In the new version in Fig. 2 (right), the new program not only shows the shortest path, but also finds the most economic path. Three update points are inserted in function `Query` from Line 24 to Line 30.



(a) Original graph (distance) (b) Updated graph (distance & cost)

Fig. 3. The shortest path before and after updating (Color figure online)

Simulating a dynamic updating scenario. Given an original C program annotated with update points, KupC can compile it with a patch file and generate binary code that is executable on $\mathbb{K}$. During execution, updating is applied once reaching a safe updating point. It simulates the behavior of a dynamic updating to a program that is running on a real-world operating system.

Figure 3 shows the results of the simulation. Figures 3(a) and (b) show the original graph and the updated graph, respectively. When the update takes place at `point1`, the output of first call is the red path in Fig. 3(a). While the second call produces two paths as shown in Fig. 3(b). The red one is the shortest path and the green one is the most economic path.

```
================================================================================
Case 1                    |  Case 2                    |  Case 3
Update at: point1         |  Update at: point1         |  Update at: point3
Output: "7km $3; 7km $3"  |  Output: "6km; 7km $3"     |  Output: "6km; 7km $3"
--------------------------------------------------------------------------------
Case 4                    |  Case 5
Update at: point3         |  Update at:
Output: "6km; 6km"        |  Output: "6km; 6km"
================================================================================
```

Fig. 4. All possible updating results searched by the state space explorer of KupC

Exploring all dynamic updating results. In addition to simulating one possible updating scenario, KupC can search for all possible updating results by exploring each possible updating point using the state space explorer.

We compile and execute the program `map` with the option `UPSEARCH=1` to invoke the state exploration function. Figure 4 shows all five different updating results. The outputs are divided into two parts by semicolon, representing the results of the two function calls of `Query`, respectively. Case 1 and Case 2 show the results when updating occurs at `point1`. Case 3 and Case 4 are for `point3`. Case 4 shows the result when updating is not performed.

While the dynamic updating occurs during the first call of the function `Query` at `point3` in Case 3, the output of the first call is not affected by updating. The reason is that the updated content will not take effect until the next access after

updating. Therefore, the outputs in Case 4 are exactly the same as the ones in Case 5. Updating at `point2` violates the safety policies. Therefore, there is no case corresponding to `point2`. All the updating results searched are valid.

Model checking temporal properties. Dynamic updating is a temporal behavior in that the properties before and after updating may be different. Such differences can be formalized as temporal properties. Another attractive function of KupC is to verify these temporal properties using LTL model checking.

As an example, we verify whether or not updating in the GPS example can be finally deployed. First, we introduce an atomic proposition called `__update`, which is false before updating and becomes true after the program is updated. Given the command `UPLTLMC = "TrueLtl ULtl __update" ./map`, KupC returns true, indicating that updating can be eventually performed.

Another property of interest is that the shortest path must become 7 after the system is updated. It can be defined as an LTL formula `__update->(<>(x==7))`, where variable x stores the value of the shortest path. Given the command `UPLTLMC="''('~Ltl__update'\'/Ltl'('TrueLtlULtl'('x==7')'')'')'"./map`, KupC returns true, indicating that updating result is correct as expected.

4 Concluding Remarks and Ongoing Work

We have presented the design and implementation of an operational semantics-based verification tool called KupC for dynamic software updating. Three case studies showed the effectiveness of KupC for the formal analysis of the dynamic software updating of C programs by simulation, state exploration, and LTL model checking. Semantics-based formalization is promising in providing effective and practical solutions for guaranteeing the correctness of dynamic software updating. For instance, Lounas *et al.* achieved formal verification of dynamic updating of Java programs based on Java's semantics [7]. Compared with their approach, our approach is more general and extendable as $\mathbb{K}$ provides an elegant semantic framework for the definition of programming languages and an easy-to-use automated verification tool generation service.

KupC is at a good position for practical code-level verification of DSU. It is directly applicable to the code and shows the feasibility of formalizing a dynamic updating mechanism on the basis of the operational semantics of target programming languages. To verify the dynamic updating of more complex and practical programs, a complete semantics of C including those of standard libraries is needed. The efficiency of KupC also needs to examine although the efficiency of $\mathbb{K}$ has been validated [9]. There is ongoing work on these directions.

KupC has some limitations because of theoretical and practical challenges in the formal verification of DSU. Theoretically, Gutpa *et al.* have shown the undecidability of the reachability of updating points [3]. Another issue is that there is no uniform definition of *correctness* of dynamic updating. The logical correctness of dynamic updating depends on target programs and its formalization relies on programmers' interpretation. Although KupC does not require

any abstraction of target programs, we suspect that certain abstraction is necessary for optimizing efficiency and scalability of the verification. For instance, a function that is not modified in a new version can be considered atomic for verification purpose. It is still an ongoing quest for an appropriate abstraction of target programs for the scalability while maintaining the validity of verification.

References

1. Duggan, D.: Type-based hot swapping of running modules. In: ICFP 2001, vol. 36, pp. 62–73. ACM (2001)
2. Ellison, C., Rosu, G.: An executable formal semantics of C with applications. In: POPL 2012. pp. 533–544. ACM (2012)
3. Gupta, D., Jalote, P., Barua, G.: A formal framework for on-line software version change. IEEE Trans. Soft. Eng. **22**(2), 120–131 (1996)
4. Hayden, C.M., Magill, S., Hicks, M., Foster, N., Foster, J.S.: Specifying and verifying the correctness of dynamic software updates. In: Joshi, R., Müller, P., Podelski, A. (eds.) VSTTE 2012. LNCS, vol. 7152, pp. 278–293. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-27705-4_22
5. Hicks, M., Nettles, S.: Dynamic software updating. ACM Trans. Prog. Lang. Syst. **27**(6), 1049–1096 (2005)
6. Hoare, C.A.R.: Record of a workshop on programming languages for distributed computing. In: Whitby-Strevens, C. (ed.) University of Warwick, p. 54 (1979)
7. Lounas, R., Mezghiche, M., Lanet, J.L.: A formal verification of dynamic updating in a Java-based embedded system. IJCCBS **7**(4), 303–340 (2017)
8. Neamtiu, I., Hicks, M., et al.: Practical dynamic software updating for C. In: PLDI 2006, pp. 72–83. ACM (2006)
9. Rosu, G.: $\mathbb{K}$: a semantic framework for programming languages and formal analysis tools. In: Dependable Software Systems Engineering, pp. 186–206. IOS Press (2017)
10. Seifzadeh, H., Abolhassani, H., Moshkenani, M.S.: A survey of dynamic software updating. J. Softw. Evol. Process **25**(5), 535–568 (2013)
11. Zhang, M., Ogata, K., Futatsugi, K.: An algebraic approach to formal analysis of dynamic software updating mechanisms. In: APSEC 2012, pp. 664–673. IEEE (2012)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.