JAIST Repository
https://dspace.jaist.ac.jp/

Title	Formalization and Verification of Behavioral Correctness of Dynamic Software Updates
Author(s)	Zhang, Min; Ogata, Kazuhiro; Futatsugi, Kokichi
Citation	Electronic Notes in Theoretical Computer Science, 294: 12-23
Issue Date	2013-03-15
Type	Journal Article
Text version	publisher
URL	http://hdl.handle.net/10119/12218
Rights	© 2013 Elsevier B.V. Open access under CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/3.0/). Min Zhang, Kazuhiro Ogata, Kokichi Futatsugi, Electronic Notes in Theoretical Computer Science, 294, 2013, 12-23, http://dx.doi.org/10.1016/j.entcs.2013.02.013
Description	

JAIST
JAPAN
ADVANCED INSTITUTE OF
SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

Available online at www.sciencedirect.com

SciVerse ScienceDirect

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 294 (2013) 12–23

www.elsevier.com/locate/entcs

Formalization and Verification of Behavioral Correctness of Dynamic Software Updates

Min Zhang, Kazuhiro Ogata, Kokichi Futatsugi

Research Center for Software Verification
Japan Advanced Institute of Science and Technology (JAIST)
Asahidai 1-1, Nomi, Ishikawa, Japan

Abstract

Dynamic Software Updating (DSU) is a technique of updating running software systems on-the-fly. Whereas there are some studies on the correctness of dynamic updating, they focus on how to deploy updates *correctly* at the code level, e.g., if procedures refer to the data of correct types. However, little attention has been paid to the correctness of the dynamic updating at the behavior level, e.g., if systems after being updated behave as expected, and if unexpected behaviors can never occur. We present an algebraic methodology of specifying dynamic updates and verifying their behavioral correctness by using off-the-shelf theorem proving and model checking tools. By theorem proving we can show that systems after being updated indeed satisfy their desired properties, and by model checking we can detect potential errors. Our methodology is general in that: (1) it can be applied to three updating models that are mainly used in current DSU systems; and (2) it is not restricted to dynamic updates for certain programming models.

Keywords: dynamic software updating, algebraic specification, verification, behavioral property

1 Introduction

Dynamic software updating (DSU) is a promising software maintenance technique for updating running software systems on the fly without incurring downtime. Like repairing a machine that is operating at full speed, dynamic updates must be highly reliable to guarantee the systems after being updated must behave as expected. That is because target systems that need dynamic updating are usually mission-critical. They need to provide 24x7 services, such as traffic control systems and financial transaction systems. Therefore, it is important to guarantee a dynamic update is correct and it can be correctly performed to the target system.

Though several studies have been conducted on the correctness of dynamic software updating, most of them consider the correctness at the code level [9,22,16],

[1] Email: {zhangmin, ogata, futatsugi}@jaist.ac.jp

1571-0661 © 2013 Elsevier B.V. Open access under CC BY-NC-ND license.
doi:10.1016/j.entcs.2013.02.013

that is, how to correctly deploy dynamic updates, e.g., whether the updated programs remain type safe [9,22] or version consistent [22,16]. Type safety of dynamic updates means that procedures will never refer to those data which have different types from their signatures. For instance, suppose that a data type `A` is updated to `A'` and there is a procedure `f(A a)` in the old version. After update `f` should refer to those data of the new type `A'` rather than those of `A`. Version consistency means no bad calls between procedures of old and new versions. There may be such a call between procedures that it happens neither in old version nor in the new one. For instance, suppose that there are procedures `f(){g();h();}`, `g(){...}`, and `h(){...}` in an old program. `f` and `g` are updated to `f(){g();}` and `g(){h();}`. Suppose that the update takes place in old `f` before `g` is called. After update, new `g` is called. New `g` calls `h`. After `g` returns, `h` is called by old `f`. Such a call should be avoided during updating.

Though code-level correctness is necessary to dynamic updating, we believe that the correctness on systems' behaviors level is equally important. Even if an update is correctly implemented at code level, such as type safe and version consistent, it may not make a system behave as expected after the update is performed. A dynamic update in Section 2 is such an example, where a system running a flawed mutual exclusion protocol is dynamically updated to a correct one. After being updated, the system is supposed to satisfy two desired properties, i.e., *mutual exclusion* and *freedom of deadlock*. However, the updated system satisfies mutual exclusion property, but may cause deadlock. Such an update is not considered behaviorally correct.

We propose an algebraic methodology of specifying dynamic updates and verifying their behavioral properties. Dynamic updates are modeled as observational transition systems (OTSs), a kind of abstract state machines that are used to specify computer systems in algebraic ways [19]. OTSs can be specified as equational theories and rewrite theories which are used for theorem proving and model checking [8,7], respectively. By theorem proving, we can show desired behavioral properties are indeed satisfied by updated systems, and by model checking we find counterexamples for those properties that do not hold. Counterexamples can help us better understand the behavior of updated systems, detect the errors of dynamic updates, and finally design correct ones.

Our method is general from the following two perspectives. Firstly, it can be used to formalize three main dynamic updating models, i.e., *interrupt model*, *invoke model*, and *relaxed consistency model* (see Section 2.1 for details), which are widely used in current dynamic software updating systems. We can formalize in the methodology the dynamic updates that conform to one of the three models. Secondly, we formalize the design of dynamic updates instead of the codes, which is different from other existing approaches e.g., the formalizations of dynamic updates in C-like programs in [12] and [13]. Hence, our formalism is not restricted to the dynamic updates that are developed in certain program models.

The rest of this paper is organized as follows. Section 2 introduces dynamic software updating. Section 3 describes the approach to formalizing dynamic updates.

Section 4 describes the verification of behavioral correctness. Section 5 discusses some related work, and Section 6 concludes the paper.

2 Dynamic Software Updating

2.1 Dynamic updating models

Dynamic software updating is different from static updating. The latter is a traditional approach to evolving software into newer versions by stopping running systems, applying updates and then restarting systems again. However, dynamic updating supports on-the-fly updates to the running systems without shutting them down.

A number of systems have been designed and implemented to support dynamically updating software systems, such as Podus [10], OPUS [1], Ginseng [17], POLUS [6]. They are specific to dynamic updates to certain class of software systems. For instance, Ginseng supports dynamic updates to single-threaded systems, and POLUS supports those to multi-threaded ones. Both only support dynamic updates to the systems developed in the C language.

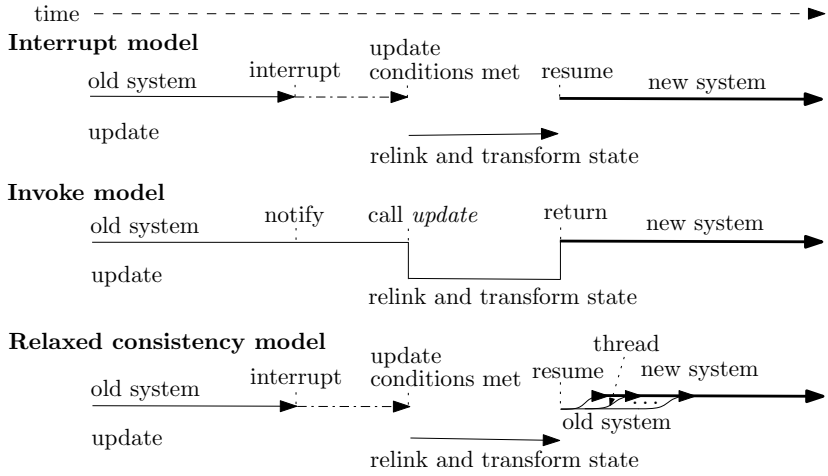

Fig. 1. Three dynamic updating models

Behind dynamic updating systems are the updating models that determine which kinds of dynamic updates they can support. Hicks et al. has grouped dynamic updating models into two groups according to the time when the updates should be applied [14]. One is called *interrupt model* and the other is *invoke* model, as visualized by the first and second portions of Fig. 1. In interrupt model, an update to a system could be started at any moment during the system's execution. The system is first interrupted at some point. Then update is performed atomically, and finally the system is resumed. The atomicity of performing the update means that the update cannot be performed in parallel with program execution. Updates are not necessarily performed at the moment of interruption. They may be delayed until certain conditions are satisfied. For instance, some dynamic updating systems forbid updates to active code (code on the stack) [5]. Some systems delay the

updates to active code until they are inactive [21]. However, the evolution to the new system always occurs immediately upon the resumption of systems. A number of dynamic updating systems are based on interrupt model, such as Podus [10], Ginseng [17], and those in [5,21,20].

Unlike the interrupt model, the invoke model requires update points to be statically specified in systems. At update points, a special *update* procedure can be invoked to perform updates. The running system is first notified that it should perform an update. It invokes the *update* procedure when reaching an update point, and the corresponding update is then performed. After the *update* procedure returns, the system continues from where it left off. Invoke model well suits the updates to multi-threaded systems. A typical example is Erlang, which supports dynamic updating based on the invoke model [3].

Relaxed consistency model is an extension of interrupt model, as visualized by the third portion in Fig. 1. It supports dynamic updates to multi-threaded systems. The difference is that relaxed consistency model allows the concurrent activity of both old and new systems and the co-existence of both old and new system states. After update is performed and the running multi-threaded system is resumed, each thread may not execute new codes immediately. Instead, it may continue to execute old code and evolve to the new code at some specific point. Update is completed when all threads evolve to the new code. Relaxed consistency model has been implemented in POLUS [6].

2.2 Behavioral correctness of DSU

Behavioral correctness means that the behavior of the systems after being updated must be correct. Formally, a dynamic update is *behaviorally correct* if the system after being updated by it satisfies its desired properties. Such properties depend on concrete systems, and usually describe the differences between the old and new systems. For instance, version 1.1.2 of the vsftpd FTP server introduced a feature that limits the number of connections from a single host. If we update a running vsftpd server of an earlier version to version 1.1.2, one of the desired properties of the server after being updated is that the number of connections from a single host will never exceed the maximal connection number.

Behavior-level correctness is equally important, compared with code-level correctness. Behavior-level correctness focuses on how the behavior of systems to be updated is affected by updating, while code-level correctness focuses on the implementation of updates. Obviously, behavior-level correctness depends on the code-level correctness, while code-level correctness is not enough to guarantee behavior-level correctness. That is, even if an update is correctly applied to a running system (no code-level errors), the system after being updated may not behave as expected.

In the rest of this section, we give an example to show the behavior-level correctness of dynamic updates. We consider a dynamic update to a system running a flawed mutual exclusion protocol (called protocol A for convenience) with a correct one (protocol B). After being updated the system executes the correct protocol. We design an update approachS, and perform it in relaxed consistency model.

The basic idea of protocol A and B is that there is a shared Boolean variable *locked* indicating whether critical section is available. Each process waits at the pre-critical section (ps) to enter the critical section (cs) until *locked* becomes false. It sets *locked* true, and enters the critical section. It sets *locked* false when it is leaving the critical section. The difference is that in protocol A checking the value of *locked* and setting *locked* true are two separate operations, while in protocol B they are treated as an atomic one. It is well-known that protocol A does not satisfy the mutual exclusion property.

We consider a dynamic update to force a system that is running protocol A to evolve to executing protocol B. The update is performed in relaxed consistency model, as depicted in Fig. 2. There are multiple processes in the old system executing protocol A. At some moment the system is interrupted, and update is performed. The value of *locked* in protocol B (denoted by $locked_B$) is assigned with the value of *locked* in protocol A (denoted by $locked_A$) in the state where interruption takes place. Then, the system is resumed. If a process is at the remainder section rs_A in protocol A, it stops executing protocol A and instead evolves to executing protocol B from the beginning rs_B. Otherwise, the process will continue to execute protocol A until it returns back to the remainder section. After all processes evolve, the update is completed. All processes run protocol B thereafter. From that moment, the system is supposed to satisfy mutual exclusion property. In Section 4, we will prove that the property indeed holds. However, we also find a counterexample showing that the system after being updated may cause deadlock. In that sense, the update cannot be considered as behaviorally correct.

3 Formalizations of dynamic updates

In formal methods, software systems are typically formalized as transition systems, consisting of a set S of states and a binary relation $\rightarrow \subseteq S \times S$ of transitions. Observational transition system (OTS) is a variant of transition systems represented in algebraic form. An OTS $\mathcal{S}$ is a triple $\langle \mathcal{O}, \mathcal{I}, \mathcal{T} \rangle$, consisting of a set $\mathcal{O}$ of observers, $\mathcal{I}$ initial states, and $\mathcal{T}$ transitions. Each state is identified by the values returned by each observer in $\mathcal{O}$. Two states v_1, v_2 are equal if each observer returns the same value from them. States in an OTS are inductively defined. Suppose that v is a

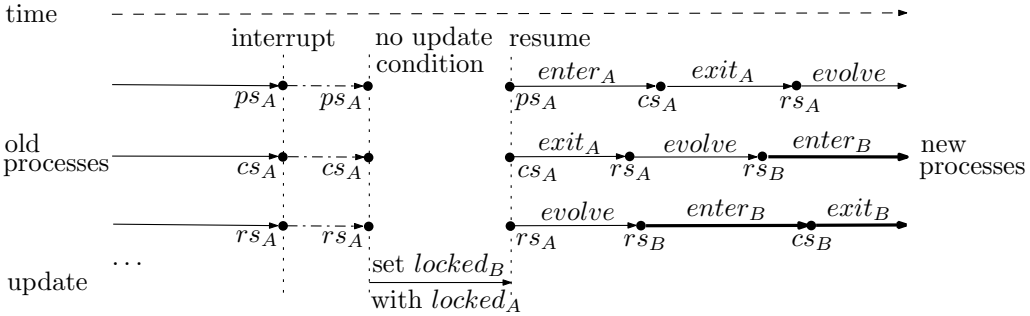

Fig. 2. A dynamic update from protocol A to B

state. For each $t \in \mathcal{T}$, $t(v, \ldots)$ is also a state, where $\ldots$ denotes other arguments. More details about the definition and applications of OTS can be referred to [19].

We view a dynamic updating to a running system as a process of evolving from an old running system to the new target system. The evolution can be divided into three steps. In interrupt model and relaxed consistency model, the three steps are: *interrupting* old system, *updating* when conditions are met, and *resuming* the system. In invoke model, they are *notifying* to invoke update, *calling update* procedure, and *returning* from the call. Generally, we consider them as *preparing*, *updating*, and *finishing*. The three steps divide the whole evolution from old system to new one into four phases, i.e., *pre-updating*, *waiting* for updating, *updated*, and *post-updating*, as depicted by Fig. 3. During *pre-updating* phase, the old system is executing. In invoke model, it continues running during *waiting* phase, but stops in other two models. Update is performed when certain conditions are satisfied in the waiting phase. After being updated, the evolution proceeds to *updated* phase, where the system waits to be resumed (in interrupt and relaxed consistency model), or for the return from *update* procedure (in invoke model). The system then goes into *post-updating* phase, where some delayed updates may be performed. Finally the system evolves to the new one.

We declare an observer to represent the four phases, and formalize the three steps by three transitions. Let U be a set of four phase names, i.e., $\{pre\text{-}updating, waiting, updated, post\text{-}updating\}$. We declare an observer *phase*, which returns the phase of the evolution in given state. The three steps are defined by three transitions *prepare*, *update*, and $finish$. We take the definition of *prepare* for example. One condition that *prepare* happens in a state v is that v is in *pre-updating* phase, i.e., $phase(v) = pre\text{-}updating$. After *prepare* happens, the evolution goes into *waiting* phase. Namely, the value of *phase* in $prepare(v)$ is $waiting$.

A dynamic update can be formalized by the union of two OTSs that represent the old and new systems with the three transitions and the observer. Suppose that the old and new systems are formalized by two OTSs $\mathcal{S}_o$ and $\mathcal{S}_n$ such that $\mathcal{S}_o = \langle \mathcal{O}_o, \mathcal{I}_o, \mathcal{T}_o \rangle$ and $\mathcal{S}_n = \langle \mathcal{O}_n, \mathcal{I}_n, \mathcal{T}_n \rangle$. An update from $\mathcal{S}_o$ to $\mathcal{S}_n$ can be formalized by $\mathcal{S}_u = \langle \mathcal{O}_u, \mathcal{I}_u, \mathcal{T}_u \rangle$, where:

- $\mathcal{O}_u \triangleq \mathcal{O}_o \cup \mathcal{O}_n \cup \{phase\}$
- $\mathcal{I}_u \triangleq \{v_0 | v_0 \in \mathcal{I}_o \wedge \boldsymbol{phase(v_0) = pre\text{-}updating}\}$
- $\mathcal{T}_u \triangleq \mathcal{T}_o \cup \mathcal{T}_n \cup \{prepare, update, finish\}$

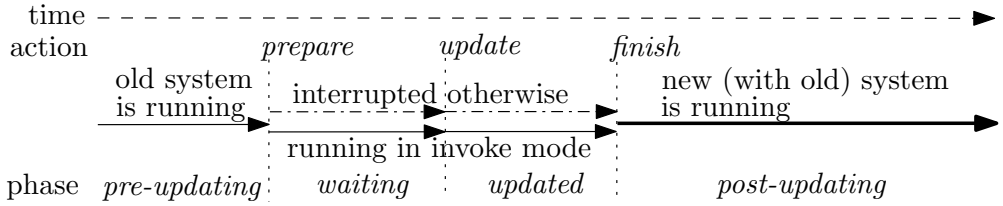

Fig. 3. Four-phase system evolution by dynamic updating

We assume that $\mathcal{O}_o \cap \mathcal{O}_n = \emptyset$ and $\mathcal{T}_o \cap \mathcal{T}_n = \emptyset$. The assumption can be achieved by renaming the observers and transitions if their intersections are not empty. Transitions in $\mathcal{T}_o$ can take place in *pre-updating* and *post-updating* phase, while those in $\mathcal{T}_n$ only take place in *post-updating* phase. Transitions in $\mathcal{T}_n$ begin from the state where the old system evolves.

We consider the formalization of the dynamic update from protocol A to B as an example. A part of the OTS is graphically shown in Fig. 4. Each circle denotes a state. The values in each state that are affected by transitions are given beside circles. For instance, $(pc_A[p] : rs)$ denotes that in the state process p is at the remainder section in protocol A. Each arrow denotes a transition with the transition name and appropriate arguments. For instance, $(enter_A, p)$ denotes process p enters the critical section. The OTS is a union of the two OTSs that formalize the protocol A and B respectively. It includes the three transitions formalizing the updating process and a transition *evolve* formalizing the evolution of each process in the *post-updating* phase.

4 Specification and Verification

We can specify OTSs and verify their desired properties by existing algebraic languages and verification systems. CafeOBJ is such a language in which OTSs are tailored to be specified [19]. CafeOBJ is also a powerful rewrite system which is often used as a proof assistant [8]. We can also translate CafeOBJ specifications of OTSs into Maude [23], a sibling algebraic language of CafeOBJ but supporting model checking [7]. we can model check the desired properties of OTSs with the generated Maude specifications.

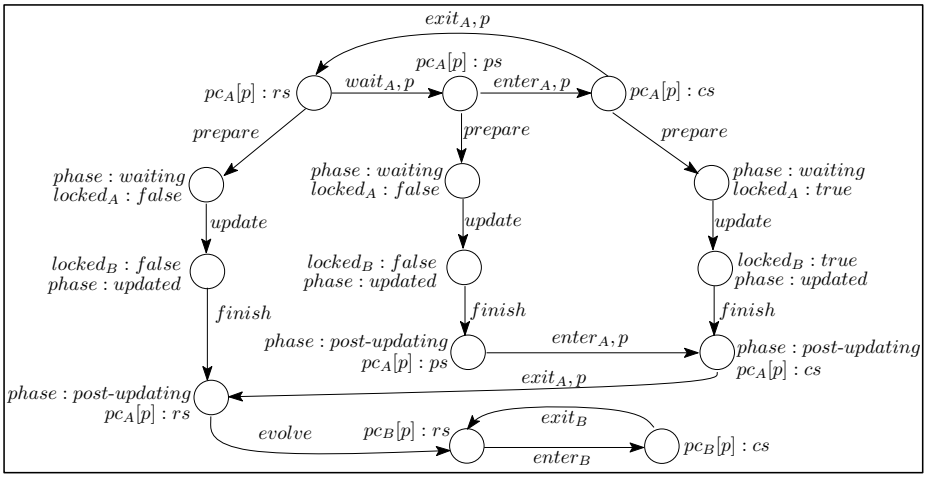

Fig. 4. A graphical OTS of the dynamic update from protocol A to B

4.1 Verification by theorem proving

In CafeOBJ, objects are classified by *sorts*. A sort denotes a kind of objects. For instance, sort `Bool` denotes a class of Boolean values *true* and *false*, which are represented by two constants `true` and `false` of `Bool`. In this paper, we use sorts `Sys`, `Pid` and `Label` to denote the classes of system states, processes, and program counters such as *rs*, *ps*, and *cs* (which are represented by constants `rs`, `ps`, and `cs` respectively).

Observers and transitions are represented by operators. For instance, transition $wait_A$ is declared by `op wait-A : Sys Pid -> Sys`, where `op` is a keyword to declare operators. The meaning of operators is defined by equations. We take the equation defined for `wait-A` and `pc-A` for example.

```
ceq pc-A(wait-A(S,I),J) = (if I = J then ps else pc-A(S,J) fi)
 if c-wait-A(S,I) .
```

`S` is a variable of `Sys`; `I` and `J` are variables of `Pid`. Keyword `ceq` is used to declare a conditional equation.

The CafeOBJ specifications of OTSs are used to prove invariant properties by proof scores [11]. We first specify the property to be proved and then compose the proof score by using CafeOBJ interactively as a proof assistant.

We prove that the system after being updated by the approach in Section 2.2 satisfies the mutual exclusion property, i.e., for any two processes p_1, p_2 in P and each state υ in Υ, $pc_B(\upsilon, p_1) = cs$ and $pc_B(\upsilon, p_2) = cs$ imply that $p_1 = p_2$. That is, in any state where there are two evolved processes at the critical section, they must be the same one. The mutual exclusion property can be represented by a predicate `mu`, which is declared and defined as follows:

```
op mu : Sys Pid Pid -> Bool
eq mu(S,I,J) = (pc-B(S,I) = cs and pc-B(S,J) = cs) implies I = J .
```

We successfully proved by CafeOBJ that the system after being updated indeed satisfies the mutual exclusion property. The proof is based on structural induction. Three lemmas are needed in the proof. We omit the details of the proof due to the limit of space.

4.2 Verification by model checking

However, some properties may fail to be satisfied by an updated system. In that situation, counterexamples of the properties are desired to refute the properties being verified. We can specify OTSs in Maude as rewrite theories and model check them with desired properties.

In Maude, states of OTSs are specified by sets of values. Each value is represented as a *component* and a set of components is called a *configuration*. For instance, each state in $\mathcal{S}_I$ consists of Boolean values of $locked_A$ and $locked_B$, a value that denotes the phase of update, the values of pc_A and pc_B for each process, and Boolean values denoting if corresponding processes have evolved. We consider an instance of the system with two processes p_1 and p_2. p_1 and p_2 are represented by two constants `p1` and `p2`

of `Pid`. A state is represented by a configuration which is of the pattern `(locked-A:□)(locked-B:□)(phase:□)(pc-A[p1]:□)(pc-A[p2]:□)(pc-B[p1]:□)(pc-B[p2]:□)(evolved[p1]:□)(evolved[p2]:□)`, where each □ holds a corresponding value in a specific state, and components are concatenated by an empty associative and commutative operator.

Transitions are represented by rewrite rules. For instance, transition *update* can be specified by the following rewrite rule:

```
rl [update] (locked-A : B) (locked-B : B') (phase : waiting) =>
            (locked-A : B) (locked-B : B)  (phase : updated) .
```

Keyword `rl` is used to declare a rule. Following `rl` is the name of the rule. `B` and `B'` are variables of `Bool`. The left-hand side is a pattern. Any configuration or a segment (a subset of components in a configuration) that matches the pattern can be rewritten by the rule. The right-hand side is the result into which the pattern is rewritten. Other transitions can be specified as rewrite rules likewise.

As an example, we use the Maude LTL (Linear Temporal Logic) model checker [7] to verify the deadlock freedom property of the system when it is updated from protocol (A) to (B). Because model checking requires the system's state space to be finite, we make it finite by fixing the number of processes in the system. We consider the case when there are only two processes in the system, which is specified in above section.

We first specify the deadlock freedom property as an LTL formula. Deadlock in the system with two processes means that both the two processes are at the remainder section when they are executing protocol (B) but the value of $lock_B$ is true. Neither of them can enter the critical section. We declare two proposition constructors `@rs?` and `locked?`. Given a process `I` and a configuration, `@rs?(I)` is true if the configuration contains `(pc-B[I]: rs)`, and false otherwise. `locked?` is true if a configuration contains `(locked-B : true)`, and false otherwise. The deadlock freedom property is defined by the formula `[]~ @rs?(p1) /\ @rs?(p2) /\ locked?`, where `[]` corresponds to the global operator **G** in LTL, `~` to ¬, and `/\` to ∧.

Maude returns a counterexample which violates the formula. The counterexample is shown in Fig. 5. Each circle represents a configuration, and an arrow denotes the name of the rule applied. We only show those components in each configuration changed by the rule. There is a path from the initial state denoted by c_0 to the state denoted by c_8, where deadlock occurs.

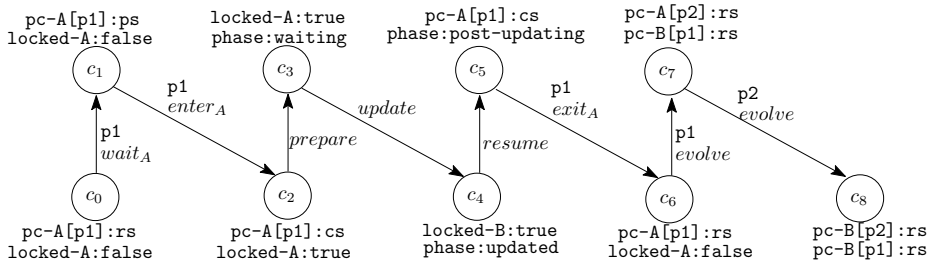

Fig. 5. A counterexample of the deadlock freedom property of the dynamic update

From counterexamples we can learn why the desired properties fail and find

possible solutions to redesign the updates. For instance, the above counterexample shows the reason why deadlock occurs is that the value of $locked_B$ is set true when update takes place. There are two possible solutions. One is to set an update point where $locked_A$ is false and perform the update in the invoke model. The other choice is to force $lock_B$ to be false, regardless of the value of $lock_A$ when the update takes place in the interrupt model.

5 Related Work

In an earlier study, Gupta et al. introduced the notion of *validity* to dynamic updates [12]. Informally, an update is called valid if a reachable state in a new system can be eventually reached from the state where the update takes place in the old system. However, the definition of validity, as being observed by Hayden et al., is both too restrictive and too permissive [13]. They proposed to define the correctness of dynamic updates by the properties that are specific to concrete dynamic updates [13]. In our work, we explicitly call the properties *behavioral properties*, to differentiate them from the code-level properties.

There are multiple approaches to the formalization of dynamic updates. In [13], dynamic updates are formalized by merging old and new versions of C programs with a merging transformation. In [12], programs are viewed as state machines. However, their approach is restricted to concrete program models, like the one in [13]. Bierman et al. proposed an update calculus to formalize dynamic software updating [4]. The calculus is flexible and independent from program models. However, they do not provide any verification support based on their formalization. Stoyle et al. proposed Proteus, a core calculus to model dynamic software updating [22]. But they use it to check code-level correctness. Most of the above formalisms except [4] are only applicable to the updates of single-threaded systems, but not to the updates of distributed systems. Anderson has provided a definition of safe dynamic updates for behavioral properties of concurrent programs [2]. However, only type safety property is considered in the definition. In our approach, we focus on the design of the old and new systems and updates, which makes our formalization approach more general but not restricted to certain programming models.

6 Conclusion and Future Work

We have introduced a notion of behavioral correctness of dynamic updates, and motivated the need of it with a concrete example. A dynamic update is called behaviorally correct if expected behavior must happen after it is performed. We proposed an algebraic methodology of specifying dynamic updates and verifying their behavioral properties. The methodology can be applied to the formalizations of three dynamic updating models, which are widely implemented in dynamic software updating systems. Unlike most of the existing formalizations of dynamic updates, our methodology is not restricted to the dynamic updates developed in certain program models. With our formalization methodology, we can verify the

behavioral properties of dynamic updates using off-the-shelf theorem proving and model checking tools. By verification, we can prove the desired properties indeed hold, or detect potential errors of dynamic updates and find possible solutions to them.

In ongoing work, we plan to apply our approach to the verification of more complicated dynamic updates. One candidate is the update from a flawed authentication protocol NSPK (Needham-Schroeder Public Key Protocol) [18] to its modified version NSLPK [15] (which has been verified to satisfy the mutual authentication property). The property of interest is that after being updated, the system should satisfy the mutual authentication property.

References

[1] Altekar, G., I. Bagrak, P. Burstein and A. Schultz, *Opus: Online patches and updates for security*, in: *14th USENIX Security*, 2005, pp. 287–302.

[2] Anderson, G. A., *Dynamic Software Update for Behavioural Properties of Concurrent Programs*, in: *Grace Hopper Celebration of Women in Computing*, 2011.

[3] Armstrong, J., R. Virding, C. Wikstr, M. Williams et al., "Concurrent programming in ERLANG," Prentice Hall, 1996.

[4] Bierman, G., M. Hicks, P. Sewell and G. Stoyle, *Formalizing dynamic software updating*, in: *2nd USE*, 2003.

[5] Boyapati, C., B. Liskov, L. Shrira et al., *Lazy modular upgrades in persistent object stores*, , **38**, ACM, 2003, pp. 403–417.

[6] Chen, H., J. Yu, C. Hang et al., *Dynamic software updating using a relaxed consistency model*, IEEE Transactions on Software Engineering (2011), pp. 679–694.

[7] Clavel, M., F. Durán and et al., *All about Maude*, LNCS 4350, Springer (2007).

[8] Diaconescu, R. and K. Futatsugi, "CafeOBJ Report," World Scientific, 1998.

[9] Duggan, D., *Type-based hot swapping of running modules*, , **36** (2001), pp. 62–73.

[10] Frieder, O. and M. Segal, *On dynamically updating a computer program: From concept to prototype*, Journal of Systems and Software **14** (1991), pp. 111–128.

[11] Futatsugi, K., *Verifying specifications with proof scores in cafeobj*, in: *21st ASE*, IEEE, 2006, pp. 3–10.

[12] Gupta, D., P. Jalote and G. Barua, *A formal framework for on-line software version change*, IEEE Transactions on Software Engineering **22** (1996), pp. 120–131.

[13] Hayden, C., S. Magill, M. Hicks et al., *Specifying and verifying the correctness of dynamic software updates*, in: *4th VSTTE, LNCS 7151*, 2012, pp. 278–293.

[14] Hicks, M. and S. Nettles, *Dynamic software updating*, ACM TOPLAS **27** (2005), pp. 1049–1096.

[15] Lowe, G., *An attack on the needham-schroeder public-key authentication protocol*, Information processing letters **56** (1995), pp. 131–133.

[16] Neamtiu, I., M. Hicks, J. Foster et al., *Contextual effects for version-consistent dynamic software updating and safe concurrent programming*, , **43**, ACM, 2008, pp. 37–49.

[17] Neamtiu, I., M. W. Hicks, G. Stoyle et al., *Practical dynamic software updating for c*, in: *PLDI* (2006), pp. 72–83.

[18] Needham, R. and M. Schroeder, *Using encryption for authentication in large networks of computers*, CACM **21** (1978), pp. 993–999.

[19] Ogata, K. and K. Futatsugi, *Proof Scores in the OTS/CafeOBJ Method*, in: *FMOODS'03, LNCS 2884* (2003), pp. 170–184.

[20] Segal, M. and O. Frieder, *On-the-fly program modification: Systems for dynamic updating*, Software, IEEE **10** (1993), pp. 53–65.

[21] Soules, C., J. Appavoo, K. Hui et al., *System support for online reconfiguration*, in: *Proceedings of the 2003 USENIX Technical Conference*, 2003, pp. 141–154.

[22] Stoyle, G., M. Hicks, G. Bierman et al., *Mutatis mutandis: safe and predictable dynamic software updating*, ACM TOPLAS **40** (2005), pp. 183–194.

[23] Zhang, M., K. Ogata and M. Nakamura, *Translation of State Machines from Equational Theories into Rewrite Theories with Tool Support*, IEICE Transactions on Information and Systems **94-D** (2011), pp. 976–988.