Verifying the Design of Dynamic Software Updating in the OTS/CafeOBJ Method

Min Zhang, Kazuhiro Ogata, and Kokichi Futatsugi

Research Center for Software Verification
Japan Advanced Institute of Science and Technology (JAIST)
{zhangmin,ogata,futatsugi}@jaist.ac.jp

Abstract. Dynamic Software Updating (DSU) is a technique for updating running software systems without incurring downtime. However, a challenging problem is how to design a correct dynamic update so that the system after being updated will run as expected instead of causing any inconsistencies or even crashes. The OTS/CafeOBJ method is an effective and practical approach to specifying and verifying the design of software. In this paper, we propose an algebraic way of specifying and verifying the design of dynamic updates in the OTS/CafeOBJ method. By verifying the design of a dynamic update, we can (1) gain a better understanding of the update, e.g., how the behavior of the running system is affected by the update, (2) identify updating points where the dynamic update can be safely applied, (3) detect potential errors, and hence (4) design a safer dynamic update.

1 Introduction

Software systems are inevitably subject to changes in order to fix bugs, or add new functionality, etc. A traditional way of deploying such changes is first shutting down a running system, then installing new version or applying patches, and finally relaunching the system. However, there are a class of systems that provide non-stoppable services such as financial transaction systems, and traffic control systems. To update such systems, dynamic software updating DSU [1] is an effective approach in which running software can be updated on the fly without being shut down and relaunched.

A challenging problem with dynamic updating is how to ensure their correctness and safety so that a system after being updated will behave as expected instead of causing any inconsistencies or even crashes in the worst case. To make sure a system can be correctly updated, it is important that the update should be correctly designed, e.g., how state and behavior are changed, at which points update can be safely applied, and what properties should be satisfied by updated systems.

Several studies have been conducted on the correctness of dynamic software updating. Duggan *et al.* proposed that dynamic update should be type safe in that functions must refer to the data of desired types [2,3]. Neamtiu *et al.* introduced the notion of *version consistency*, meaning that the calls between functions in different versions should be consistent [4]. These properties are necessary to make sure that dynamic update is correctly performed at the code leve, but not sufficient. A correct dynamic update also depends upon its logical design, such as how an old system's state and behavior are changed, and when update should be applied. Gupta *et al.* proposed a formal framework to analyze the *validity* of dynamic update [5] by studying whether a reachable state of the new system can be finally reached by the updated system. However, it is generally undecidable to check the validity of an update. Hayden *et al.* proposed an approach to analyzing how the behavior of a system is affected by an update and whether the change of the behavior satisfies the requirement [6]. Their approach is designed for the updates in C programs.

Little attention has been paid to the correctness of dynamic updating at the design level. Our previous work has shown that a correct design of an update is equally important to its implementation, and proposed an approach to formalizing dynamic updating based on three updating models called *invoke model*, *interrupt model*, and *relaxed consistency model* [7]. In this paper, we classify dynamic updates into two classes, i.e., *instantaneous updating model*, and *incremental updating model* in terms of how state and behavior of old system is changed by update. We propose an approach to formalizing dynamic updates that conform to either of the two models in the OTS/CafeOBJ method. The OTS/CafeOBJ method is an algebraic approach to formalizing and verifying the design of software systems [8,9]. We choose the OTS/CafeOBJ method for its flexibility in formalization such as the support of user-defined abstract data types, and systematic verification approaches, i.e., by compositionally writing proof scores and *searching* (or model checking) [10,11]. It also supports the formalization and verification of infinite-state systems. Several case studies have been conducted to demonstrate its effectiveness [12,13,14,15].

By verifying the design of dynamic updates, we can gain a better understanding of the design, identify updating points where the dynamic update can be safely applied, detect potential errors in update, and hence design a safer one. To demonstrate the feasibility of our approach, we formalize and verify a system, which is dynamically updated from a flawed mutual exclusion protocol to a correct one. By verification, we find the update may cause the system into a deadlock state. The verification result helps us find the problem that causes the deadlock. Compared with other existing approaches [5,6], our approach is more general in that it is not specific to some concrete dynamic updates and how they are implemented. Those that conform to the instantaneous or incremental updating model can be specified and verified in this approach. Moreover, it is not specific to dynamic updates that are designed and implemented in certain programming languages.

The rest of this paper is organized as follows. Section 2 presents the two models of dynamic updating. Section 3 briefly introduces the OTS/CafeOBJ method. Section 4 describes our approach to formalizing dynamic updates in the OTS/CafeOBJ method. Section 5 shows a demonstrating example. Section 6 discusses some related work, and Section 7 concludes the paper.

2 Models of Dynamic Software Updating

We classify dynamic updates into two classes according to how they change the behavior and state of running systems. In one model, a running system may be interrupted by a DSU system first before being updated. After update, it is resumed from where it is interrupted to run the new-version code. In the other model, there can be a period during which both the old system and the new system run in parallel. After update is completed, the old system stops, while the new system keeps running. We call them *instantaneous updating model* and *incremental updating model*. In this section, we describe the two models and introduce some typical DSU systems that conform to either of them.

2.1 Instantaneous Updating Model

In instantaneous updating model, a system that is running an old version is first temporarily interrupted in some state when updating condition is satisfied. Updating is then applied, e.g., loading new-version code into memory, and converting the current old state into a corresponding new one that is consistent the new version. After updating is completed, the system is resumed from the generated state to run the new-version code. Thus, the system will behave as a new system.

Instantaneous updating model is quite suited to dynamic updating to single-threaded applications, such as web servers *vsftpd* (a commonly used FTP daemon) and *sshd* (secure shell daemon). Many DSU systems support instantaneous dynamic updating. Gupta *et al.* proposed a way of dynamic updating by using state transfer [16]. They first suspend the running process, copy it to a new one using state transfer, and upgrade it with new code. Hicks *et al.* proposed a framework for dynamic update based on patches and state transformation [1]. Neamtiu *et al.* developed a tool called Ginseng for the dynamic update to single-threaded C programs [17]. Although these DSU systems are different in terms of their ways of implementing dynamic updates, the ways of how the behavior of systems is affected by updating are the same. Namely, dynamic updates supported by these DSU systems conform to the instantaneous updating model.

2.2 Incremental Updating Model

In incremental updating model, an old system is gradually updated to running the new-version code. There is a period during which the old system and the new one can run in parallel. After updating starts, the old system keeps running, and a part of the old system can be separately updated once it satisfies the specified updating condition. The updated part will start to run the new-version code, with the other part of the system is still running the old-version code. After all the system is updated, updating is completed, and the whole system behaves like a new-version system.

Incremental updating model is well suited to dynamic updates of multi-threaded programs and distributed systems. For instance, when updating a multi-threaded program, there may be some threads that cannot be updated when update is started, e.g., because they are occupying some critical resources. These threads have to keep running the old version until they reach some state where updating condition is satisfied. After all threads are updated, the updating process is completed, and the whole system executes the new version. Updating distributed systems can be considered similar to updating multi-threaded programs by viewing each node in systems as thread.

There are some typical DSU systems supporting incremental updating. For instance, POLUS [18] is designed and implemented for dynamic updating to multi-threaded C programs. In POLUS, they proposed a relaxed consistency model for dynamic updating. It allows the parallel execution of both the old version and new version after updating starts. The states of the two versions are bidirectionally transformable. The consistency between the two versions is ensured by the bidirectional write-through synchronization. Podus is suited to dynamic updating to distributed systems [19]. Kitsune [20] is a general-purpose dynamic updating tool for both single- and multi-threaded C applications. One common feature of them is that they all allow the co-existence of the states of old and new versions during updating, which is different from the instantaneous updating model. During updating, a system is partially updated when updating condition is satisfied. Updating is gradually completed when all parts in the system are updated.

3 The OTS/CafeOBJ Method

The OTS/CafeOBJ method is an algebraic way of formalizing, specifying and verifying the design of software systems [8,9]. OTS is abbreviated for observational transition system, a mathematical model of state transition systems. CafeOBJ is an executable algebraic specification language [21], which is well suited to specify OTS. The basic idea of the OTS/CafeOBJ method is modeling a software system as an OTS, specifying the OTS in CafeOBJ, and verifying system's properties using CafeOBJ's theorem proving or searching facilities.

3.1 Observational Transition System (OTS)

In OTS, abstract data types are used to formalize values such as natural numbers, Boolean values, and strings in software systems. System's states are characterized by the values that are returned by a special class of functions called *observers*, unlike traditional state transition systems where states are represented as sets of variables. Transitions between states are also specified by functions which we call *transitions* to differ them from ordinary functions.

We suppose that all abstract data types have been predefined for the values used in a system and denote them by D with different subscripts. Let Υ denote a universal state space.

Definition 1 (OTSs). *An OTS $\mathcal{S}$ is a triple $\langle \mathcal{O}, \mathcal{I}, \mathcal{T} \rangle$ such that:*

- *$\mathcal{O}$: A set of observers. Each observer is a function $o : \Upsilon \times D_{o1} \times \ldots \times D_{om} \to D_o$. Two states υ_1, υ_2 are equal (denoted by $\upsilon_1 =_{\mathcal{S}} \upsilon_2$) if each observer returns the same value with the same arguments from the two states.*
- *$\mathcal{I}$: A set of initial states s.t. $\mathcal{I} \subseteq \Upsilon$.*
- *$\mathcal{T}$: A set of transitions. Each transition is a function $t : \Upsilon \times D_{t1} \times \ldots \times D_{tn} \to \Upsilon$. Each t preserves the equivalence between two states in that if $\upsilon_1 =_{\mathcal{S}} \upsilon_2$, then for each $y_i (i = 1, \ldots, n)$ in D_{ti}, $t(\upsilon_1, y_1, \ldots, y_n) =_{\mathcal{S}} t(\upsilon_2, y_1, \ldots, y_n)$. Each t has an effective condition $c\text{-}t : \Upsilon \times D_{t1} \times \ldots \times D_{tn} \to \mathcal{B}$, s.t. for any state υ if $\neg c\text{-}t(\upsilon, y_1, \ldots, y_n)$, $t(\upsilon, y_1, \ldots, y_n) =_{\mathcal{S}} \upsilon$.*

OTSs can be specified in CafeOBJ as equational specifications. Each equation defined for initial states is in the form of:

`eq` $o(\upsilon_0, x_1, \ldots, x_m) = T[x_1, \ldots, x_m]$.

Keyword `eq` is used to declare an equation in CafeOBJ. The above equation is defined for an observer in the form of $o : \Upsilon \times D_{o1} \times \ldots \times D_{om} \to D_o$, where υ_0, $x_j (j = 1, \ldots, m)$ are variables of Υ and D_{oj} respectively. T is a term of D_o, representing the value observed by o with arguments $x_1, \ldots, x_m$ in all initial states.

Each equation defined for an observer $o : \Upsilon \times D_{o1} \times \ldots \times D_{om} \to D_o$ and a transition $t : \Upsilon \times D_{t1} \ldots \times D_{tn} \to \Upsilon$ is in the following form:

`ceq` $o(t(\upsilon, y_1, \ldots, y_n), x_1, \ldots, x_m) = T[\upsilon, y_1, \ldots, y_n, x_1, \ldots, x_m]$
 `if` $c\text{-}t(\upsilon, y_1, \ldots, y_n)$.

Keyword `ceq` is used to declare a conditional equation. The equation specifies all the values observed by o in the state $t(\upsilon, y_1, \ldots, y_n)$, where $y_i (j = 1, \ldots, n)$ is a variable of D_{ti}. The condition part is the effective condition of t, which says if the effective condition holds, the values observed by o in the state $t(\upsilon, y_1, \ldots, y_n)$ are equal to those represented by the term T. If the effective condition does not hold, the state $t(\upsilon, y_1, \ldots, y_n)$ is equal to υ, which is formalized by the following equation:

`ceq` $t(\upsilon, y_1, \ldots, y_n) = \upsilon$ `if not` $c\text{-}t(\upsilon, y_1, \ldots, y_n)$.

3.2 Verification in the OTS/CafeOBJ Method

Generally, there are two ways of verifying systems' properties in the OTS/CafeOBJ method. One is by theorem proving and the other is by searching (or model checking).

Verification by Theorem Proving. The basic idea of verification by theorem proving in CafeOBJ is to construct proof scores for an invariant property by using CafeOBJ as a proof assistant. Proof scores are instructions that can be executed in CafeOBJ. Verifying a system's property is actually a process of writing proof scores with humans creating the proof plan in which proof should be performed and CafeOBJ evaluating proof scores based on the proof plan. If proof scores are successfully completed and they are evaluated by CafeOBJ as expected, a desired property is proved.

The strategy of constructing proof scores is by structural induction on system states and case analysis. In the base case, we check whether the property being proved holds in the initial states defined in an OTS. If it holds, we continue to deal with the induction case. Otherwise, the proof fails. In the induction case, we make a hypothesis that the property being proved holds for a state υ, and check whether it holds for all possible successor states of υ. If it is true, the proof is finished, and otherwise fails. During proving, we may need to prove some lemmas, which are necessary to prove the main property. Interested readers can refer to [22] for more details of how to construct proof scores in the OTS/CafeOBJ method.

Verification by Searching (or Model Checking). Searching is another way of verifying invariant properties in CafeOBJ. By searching, CafeOBJ traverses the states (or a bounded number of states if the states are infinite) that are reachable from a given initial state, and check which states satisfy a specific condition. The condition is the negation of the property in order to find counterexamples of it. Once CafeOBJ returns a solution, it means that there exists an execution path from an initial state to a state where the property does not hold, which is considered as a counterexample for the failure of the property.

Searching in CafeOBJ is an effective way to find counterexamples, and particularly useful when the size of system's states is reasonably small. A more efficient searching functionality is implemented in Maude [23], a sibling language of CafeOBJ. Besides searching, Maude also provides model checking facilities which are more efficient to find counterexamples of invariant properties and even liveness properties. An OTS can also be specified in Maude, so that we can use Maude's searching and model checking facilities. Some approaches have been proposed to automatically translate a CafeOBJ specification that specifies an OTS into Maude for the same purpose [24,25].

Instead of using either theorem proving or searching (or model checking) for verification, the combination of them is also useful. During constructing proof scores for an invariant property, we can immediately stop proving once we find a counterexample for it or for some lemma which is necessary to prove the property. An approach called induction-guided falsification (IGF) has been proposed to combine theorem proving and model checking based on modeling a system as an OTS (see [26] for the details).

4 Formalization of Dynamic Updating by OTS

To design a dynamic update, we should consider five factors, i.e., the old system S_{old} which is running and waiting for updating, the new system S_{new} which will run after updating, updating model (instantaneous or incremental), updating condition φ specifying under which condition update can be applied, and a state transformation function f which is used to convert an old-version state into a new-version state.

To formalize an update, we assume that both the old and new systems S_{old} and S_{new} have been formalized by two OTSs, i.e., $\mathcal{S}_{\text{old}} = \langle \mathcal{O}_{\text{old}}, \mathcal{I}_{\text{old}}, \mathcal{T}_{\text{old}} \rangle$, and $\mathcal{S}_{\text{new}} = \langle \mathcal{O}_{\text{new}}, \mathcal{I}_{\text{new}}, \mathcal{T}_{\text{new}} \rangle$.

4.1 Formalization of Instantaneous Updating

Suppose that there is an instantaneous update designed to update S_{old} by S_{new}. We define an OTS $\mathcal{S}_{\text{ins}}$ with $\mathcal{S}_{\text{old}}$ and $\mathcal{S}_{\text{new}}$ to formalize an instantaneous update. $\mathcal{S}_{\text{ins}}$ is defined as follows:

Definition 2 (OTS $\mathcal{S}_{\text{ins}}$ of instantaneous update). $\mathcal{S}_{\text{ins}} = \langle \mathcal{O}_{\text{ins}}, \mathcal{I}_{\text{ins}}, \mathcal{T}_{ins} \rangle$*:*

- $\mathcal{O}_{\text{ins}} = \mathcal{O}_{\text{old}} \uplus \mathcal{O}_{\text{new}} \cup \{status : \Upsilon \to \mathcal{B}\}$
- $\mathcal{I}_{\text{ins}} = \{\upsilon_0 | status(\upsilon_0) = false, \upsilon_0 \in \mathcal{I}_{\text{old}}\}$
- $\mathcal{T}_{\text{ins}} = \mathcal{T}_{\text{old}} \uplus \mathcal{T}_{\text{new}} \cup \{update : \Upsilon \to \Upsilon\}$.

Operator $\uplus$ denotes a disjoint union of two sets, e.g., $\mathcal{O}_{\text{old}} \uplus \mathcal{O}_{\text{new}} = \{(o, v) | v \in \{\text{old}, \text{new}\}, o \in \mathcal{O}_v\}$. In the paper we write o_v instead of (o, v) for convenience. $\mathcal{O}_{\text{ins}}$ is a disjoint union of $\mathcal{O}_{\text{old}}$ and $\mathcal{O}_{\text{new}}$, plus a new observer *status*. We view the old system, the new system, and updating from the old system to the new one as a whole system. In that sense, updating can be considered as an internal adapting process in adaptive program [27]. We call the states of the whole system *super states*, each of which consists of an old state and a new state, plus a flag indicating whether the old system is updated or not. In $\mathcal{O}_{\text{ins}}$, observers that are from $\mathcal{O}_{\text{old}}$ are used to represent the states in old system, and those from $\mathcal{O}_{\text{new}}$ represent the states in new system. We use the observer *status* to represent the status of update. It returns false in a given state if it is a state before update, and otherwise true.

Initial states υ_0 in $\mathcal{I}_{\text{ins}}$ are those initial states in $\mathcal{I}_{\text{old}}$ and their *status* is false, i.e., $status(\upsilon_0) = false$, indicating that update cannot take place before the old system starts. Initial states of the new system are undefined in the initial states in I_{ins}. That is because new system runs from a state which is transformed from an old state where update takes place. Thus, the initial states of the new system do not affect how a system is updated.

Set $\mathcal{T}_{\text{ins}}$ includes both the transitions in the old system and those in the new one. We introduce a new transition *update* to formalize the behavior of updating from S_{old} to S_{new}. The effective condition of *update* is represented by a state predicate *c-update* : $\Upsilon \to \mathcal{B}$. We assume that the updating condition is represented by a state predicate φ, which returns true for a given old state when the updating condition is satisfied, and otherwise false. We enhance φ. Given a super state υ, φ returns true if it is true for the old state in υ, and otherwise false. The effective condition of *update* can be defined by the following equation:

$$\texttt{eq}\ \ \textit{c-update}(\upsilon) = (\texttt{not}\ status(\upsilon))\ \texttt{and}\ \varphi(\upsilon)\quad .$$

The equation says that the effective condition holds in those states before updating where the updating condition is satisfied. Transition *update* only takes affect in those states where *c-update* is true.

After *update* takes affect in a super state υ, the old state in υ is transformed into a new one. It is equal to say that the new state in υ is initialized according to the old state and transformation function. The transformation can be formalized as a set of equations. Each of them is defined for an observer in o_{new} in the following from:

$$\texttt{ceq}\quad o_{\text{new}}(\mathit{update}(\upsilon), x_1, \ldots, x_m) = T \ \texttt{ if } \mathit{c\text{-}update}(\upsilon) \quad .$$

The equation specifies the values observed by o_{new} in the state $\mathit{update}(\upsilon)$. Given parameters $x_1, \ldots, x_m$, the left-hand term represents the value observed by o_{new} in state $\mathit{update}(\upsilon)$ with respect to $x_1, \ldots, x_m$. The value equals the one of term T which usually contains the values in the old state in υ. Especially, some values in the new state are copies of the corresponding values in the old state. In that case, we use the following equation to specify the new values:

$$\texttt{ceq}\quad o_{\text{new}}(\mathit{update}(\upsilon), x_1, \ldots, x_m) = o_{\text{old}}(\upsilon, x_1, \ldots, x_m) \ \texttt{ if } \mathit{c\text{-}update}(\upsilon) \quad .$$

The values in the old state in υ are not affected by updating. Thus, old values in the state $\mathit{update}(\upsilon)$ are the same as those in υ. Thus, the values observed by each observer o_{old} are not changed. They can be defined by the following equation:

$$\texttt{eq}\quad o_{\text{old}}(\mathit{update}(\upsilon), x_1, \ldots, x_m) = o_{\text{old}}(\upsilon, x_1, \ldots, x_m) \quad .$$

Note that the above equation is unconditional. That is because no matter the effective condition holds or not in υ, the values in the old state in υ are not affected by transition *update*.

Update only takes place once in instantaneous updating model. Thus, after an update takes affect the status of the states afterwards is set true to indicate update has taken place. The following equation specifies the change of status when update takes affect:

$$\texttt{ceq}\quad \mathit{status}(\mathit{update}(\upsilon)) = \mathit{true} \ \texttt{ if } \ \mathit{c\text{-}update}(\upsilon) \quad .$$

State status is only affected by update. Transitions in both old and new systems do not change the status. For each transition $t : \Upsilon \times D_1 \ldots \times D_n \to \Upsilon$ in $\mathcal{T}_\upsilon$, we have the following equation:

$$\texttt{eq}\quad \mathit{status}(t(\upsilon, y_1, \ldots, y_n)) = \mathit{status}(\upsilon) \quad .$$

If the effective condition is not satisfied by a state υ, e.g., υ is a state after updating, or the updating condition φ is not satisfied by υ, transition *update* takes no effect on υ. This fact is specified by the following equation:

$$\texttt{ceq}\quad \mathit{update}(\upsilon) = \upsilon \ \texttt{ if not } \mathit{c\text{-}update}(\upsilon) \quad .$$

The definition of each transition that is from $\mathcal{T}_{\text{old}}$ or $\mathcal{T}_{\text{new}}$ should be slightly revised in $\mathcal{T}_{\text{ins}}$. That is because transitions from $\mathcal{T}_{\text{old}}$ only take effect before updating, while those from $\mathcal{T}_{\text{new}}$ take effect after updating. Thus, $\mathit{status}(\upsilon) = \mathit{false}$ should be a part of the effective conditions for the transitions from $\mathcal{T}_{\text{old}}$ in state υ. Similarly, $\mathit{status}(\upsilon) = \mathit{true}$ should be a part of the effective conditions for the transitions from $\mathcal{T}_{\text{new}}$. We also need to specify the facts that the values in the old state are not affected by any transitions from $\mathcal{T}_{\text{new}}$, and similarly those in the new state are not affected by the transitions from $\mathcal{T}_{\text{old}}$. Therefore, for each observer o_{old} and a transition t_{new}, they satisfy the following equation:

$$\texttt{eq}\quad o_{\text{old}}(t_{\text{new}}(\upsilon, y_1, \ldots, y_n), x_1, \ldots, x_m) = o_{\text{old}}(\upsilon, x_1, \ldots, x_m) \quad .$$

Similarly, for each observer o_{new} and transition t_{new}, they satisfy the equation:

$$\texttt{eq}\quad o_{\text{new}}(t_{\text{old}}(\upsilon, y'_1, \ldots, y'_{n'}), x'_1, \ldots, x'_m) = o_{\text{new}}(\upsilon, x'_1, \ldots, x'_{m'}) \quad .$$

In this way, we define an OTS $\mathcal{S}_{\text{ins}}$ that specifies an instantaneous update from S_{old} to S_{new}.

4.2 Formalization of Incremental Updating

Incremental updates allow concurrent execution of both old and new systems, which makes the formalization more complicated than that of instantaneous updates. During the current execution, an old state is gradually transformed into new one. Each transformation may change a fragment of old state. After all fragments of the old state are transformed, an update is completed. We divide an old state of system S_{old} into a set of *sub-states*. Each sub-state is an updating unit, meaning that a sub-state is either completely transformed into new one or completely not transformed.

We make some assumptions on the old systems in order to formalize dynamic update on it. We assume that in an old state there is a sub-set of such sub-states they have the same data fields in an old state. To differentiate such sub-states, let P be a set of index, and each sub-state is indexed with an element in P. We called them *indexed* sub-states. We further assume that other sub-states that are not in the sub-set must be transformed at the same time. We call them *unindexed* sub-states. These assumptions are reasonable for the dynamic updates on multi-threaded software systems or distributed systems. Each indexed sub-state represents the state of a thread or node, while unindexed sub-states represent shared values and resources in systems.

Suppose that an incremental update is designed for an old system S_{old} to make it updated to a new one S_{new}. We define an OTS $\mathcal{S}_{\text{inc}}$ to formalize incremental updates from S_{old} to S_{new}. We further suppose that $\mathcal{S}_{\text{old}}$ and $\mathcal{S}_{\text{new}}$ are two OTSs modeling S_{old} and S_{new} respectively.

Definition 3 (OTS $\mathcal{S}_{\text{inc}}$ of incremental update). $\mathcal{S}_{\text{inc}} = \langle \mathcal{O}_{\text{inc}}, \mathcal{I}_{\text{inc}}, \mathcal{T}_{\text{inc}} \rangle$*:*

- $\mathcal{O}_{\text{inc}} = \mathcal{O}_{\text{old}} \uplus \mathcal{O}_{\text{new}} \cup \mathcal{O}'$
- $\mathcal{I}_{\text{inc}} = \{\upsilon_0 | \upsilon_0 \in \mathcal{I}_{\text{old}}, \neg \mathit{started}(\upsilon_0), \neg \mathit{updated}(\upsilon_0, p), \neg \mathit{updated}'(\upsilon_0)\}$

– $\mathcal{T}_{\text{inc}} = \mathcal{T}_{\text{old}} \uplus \mathcal{T}_{\text{new}} \cup \mathcal{T}'$

where, $\mathcal{O}' = \{started : \Upsilon \to \mathcal{B}, updated : \Upsilon \times P \to \mathcal{B}, updated' : \Upsilon \to \mathcal{B}\}$*, and* $\mathcal{T}' = \{start : \Upsilon \to \Upsilon, update' : \Upsilon \to \Upsilon, update : \Upsilon \times P \to \Upsilon\}$.

$\mathcal{O}_{\text{inc}}$ consists of the observers in $\mathcal{O}_{\text{old}}$ and $\mathcal{O}_{\text{new}}$, and three new observers in $\mathcal{O}'$, i.e., *started*, *updated* and *updated′*. Observer *started* returns a Boolean value for a given state, representing whether an update has been started or not. Observer *updated* returns a Boolean value for a given indexed sub-state to indicate whether the sub-state is updated or not. Observer *updated′* returns true if sub-states are updated by *update′* in υ, and otherwise false. We assume that updates do not start from initial states of the old system, i.e., $started(\upsilon_0) = false$, and each sub-state is not yet updated, i.e., $updated(\upsilon_0, i) = false$ for each $i \in P$, and $updated(\upsilon_0) = false$.

The set $\mathcal{T}_{\text{inc}}$ is a disjoint union of $\mathcal{T}_{\text{old}}$ and $\mathcal{T}_{\text{new}}$, plus four transitions in $\mathcal{T}'$. Transition *start* specifies the starting of an incremental update. One of the condition of *start* is that in the current state updating must have not been started. There may be some other conditions in concrete systems to start updating. Such conditions should also be specified as part of the effective condition of *start*.

Next, we explain transitions *update* and *update′*, which formalize the updating of indexed and unindexed sub-states respectively. Suppose that there is an updating condition of updating indexed sub-states. We use a state predicate $\varphi : \Upsilon \times P \to \mathcal{B}$ to specify the condition. Namely, φ returns true for a given super state υ and a sub-state whose identifier is i if the sub-state satisfies the condition, and otherwise false. The effective condition of transition *update* can be specified by the following equation:

`eq` $c\text{-}update(\upsilon, i) = \varphi(\upsilon, i)$ `and not` $updated(\upsilon, i)$.

By updating, an indexed sub-state is transformed into new corresponding ones. We define a set of equations to specify the transformation. Each is defined for an observer in $\mathcal{O}_{\text{new}}$ whose observed value is initialized by the transformation. The equation is of the following form:

`ceq` $o_{\text{new}}(update(\upsilon, i), x_1, \ldots, x_m) = T$ `if` $c\text{-}update(\upsilon, i)$.

The above equation says in the state $update(\upsilon, i)$, the values observed by o_{new} equals T, where T is a term representing the relation between the values observed by o_{new} and some values in the old state.

Transition *update′* can be defined likewise. We assume that there is an updating condition of unindexed sub-states according to the design of the update, and we define a state predicate $\varphi' : \Upsilon \to \mathcal{B}$ to specify the condition. The effective condition of transition *update′* can be specified by the following equation:

`eq` $c\text{-}update'(\upsilon) = \varphi'(\upsilon)$ `and not` $updated'(\upsilon)$.

Unindexed sub-states are transformed at the same time into new corresponding ones. We define a set of equations to specify the transformation. Each is defined for an observer in $\mathcal{O}_{\text{new}}$ whose observed value is initialized by the transformation.

Since it is similar to the equations defined for *update*, we omit the details in the paper.

The definitions of transitions in $\mathcal{T}_{\text{inc}}$ that are from $\mathcal{T}_{\text{old}}$ and $\mathcal{T}_{\text{new}}$ need slight revision, like the formalization of instantaneous updates.

5 A Demonstrating Example

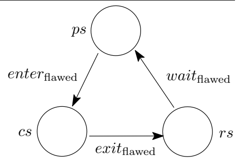
In this section, we use an example to demonstrate how to use the proposed approach to formalize and verify a concrete dynamic updating. We assume that a system is running a flawed mutual exclusion protocol, and it is dynamically updated with a correct one. We design a dynamic update that conforms to the incremental updating model. By verification, we found that the system after being updated satisfies mutual exclusion property. However, we also found it may go to a deadlock state. A counterexample is found. By analyzing the counterexample, we give two solutions to solve the deadlock problem by modifying the update.

5.1 An Update of a Mutual Exclusion Protocol

First, we explain the flawed mutual protocol, which is being executed by the running system. The pseudo-code of the protocol is shown as follows:

A flawed mutual exclusion protocol and its state transition diagram

```
Loop "remainder section"
  $rs$: repeat until $locked = false$;
  $ps$: $locked := true$;
        "critical section"
  $cs$: $locked := false$;
```

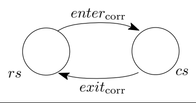


Initially, each process is at the remainder section (rs). A process waits at the pre-critical section (ps) to enter the critical section (cs) until *locked* becomes false. It sets *locked* true, and enters the critical section. It sets *locked* false when it is leaving the critical section.

The protocol does not satisfy mutual exclusion because of the non-atomicity of the action of setting *locked* true and entering the critical section. It can be solved by using an atomic operation $fetch\&store$, which takes a variable x and a value d, and atomically sets x to d and returns the previous value of x. The revised protocol and the behavior of each process in it are depicted as follows:

A correct mutual exclusion protocol and its state transition diagram

```
Loop "remainder section"
  $rs$: repeat while $fetch\&store(locked, true)$;
        "critical section"
  $cs$: $locked := false$;
```



When *locked* is false, a process atomically sets *locked* true and enters the critical section. The revised protocol has been proved to satisfy mutual exclusion in OTS/CafeOBJ method. We omit the details of the proof in the paper.

5.2 A Dynamic Updating and Its Formalization

We consider dynamically updating the system that is running the flawed mutual exclusion protocol to run the correct one. The update is performed in the incremental updating model, and can be started at any moment. When update takes place, the value of *locked* in the correct protocol is initialized with the one of the flawed protocol. After update starts, an old process will switch to executing the new protocol once it is at the remainder section. If an old process is not in the remainder section, it has to continue to execute the flawed protocol until it returns back to the remainder section. The updating is completed after all processes are updated to the correct protocol.

To formalize the update, we first formalize the flawed protocol and the correct one as OTSs $\mathcal{S}_{\text{flawed}}$ and $\mathcal{S}_{\text{corr}}$ respectively. Let L be the set $\{rs, ps, cs\}$, and P be a set of processes' identifiers. The definition of $\mathcal{S}_{\text{flawed}}$ is as follows:

Definition of OTS $\mathcal{S}_{\text{flawed}}$ for the flawed mutual exclusion protocol: $\mathcal{S}_{\text{flawed}} \triangleq \langle \mathcal{O}_{\text{flawed}}, \mathcal{I}_{\text{flawed}}, \mathcal{T}_{\text{flawed}} \rangle$.

- $\mathcal{O}_{\text{flawed}} \triangleq \{pc_{\text{flawed}} : \Upsilon \times P \to L, locked_{\text{flawed}} : \Upsilon \to \mathcal{B}\}$
- $\mathcal{I}_{\text{flawed}} \triangleq \{\upsilon_0 | pc_{\text{flawed}}(\upsilon_0, p) = rs_{\text{flawed}} \wedge \neg locked_{\text{flawed}}(\upsilon_0)\}$
- $\mathcal{T}_{\text{flawed}} \triangleq \{wait_{\text{flawed}} : \Upsilon \times P \to \Upsilon, enter_{\text{flawed}} : \Upsilon \times P \to \Upsilon, exit_{\text{flawed}} : \Upsilon \times P \to \Upsilon\}$

There are two observers pc_{flawed} and $locked_{\text{flawed}}$. Given a state υ and a process identifier p, $pc_{\text{flawed}}(\upsilon, p)$ returns a value in L, indicating the location of process p in υ, and $locked_{\text{flawed}}(\upsilon)$ represents the value of the shared Boolean variable *locked* in υ. $\mathcal{I}_{\text{flawed}}$ define the set of initial states, where all processes are at the remainder section, and *locked* is false. Three transitions are used to specify the corresponding three actions of each process, i.e., waiting for, entering and leaving the critical section. We only take $enter_{\text{flawed}}$ for example to explain the definitions of the transitions.

The effective condition is represented by a state predicate $c\text{-}enter_{\text{flawed}} : \Upsilon \times P \to \mathcal{B}$. A process p can enter the critical section if it is at the ps, which can be specified by the following equation:

`eq` $c\text{-}enter_{\text{flawed}}(\upsilon, p) = (pc_{\text{flawed}}(\upsilon, p) = ps)$.

When $c\text{-}enter_{\text{flawed}}(\upsilon, p)$ is true, p enters the critical section, and *locked* is set *true* in the state $enter_{\text{flawed}}(\upsilon, p)$. The changes are specified by the following two equations:

`ceq` $pc_{\text{flawed}}(enter_{\text{flawed}}(\upsilon, p), p') = ($`if` $p = p'$ `then` cs `else` $pc_{\text{flawed}}(\upsilon, p')$ `fi`$)$
  `if` $c\text{-}enter_{\text{flawed}}(\upsilon, p)$.
`ceq` $locked_{\text{flawed}}(enter_{\text{flawed}}(\upsilon, p)) = true$ `if` $c\text{-}enter_{\text{flawed}}(\upsilon, p)$.

In the above equation p' is a variable of P, representing an arbitrary process. It can be the same as p or others. The equation says that only p's location is changed by the transition from υ to $enter_{\text{flawed}}(\upsilon, p)$. The other two transitions can be defined likewise.

The definition of $\mathcal{S}_{\text{corr}}$ is similar to $\mathcal{S}_{\text{flawed}}$, except that we only need two transitions $enter_{\text{corr}}$ and $exit_{\text{corr}}$ to formalize process's behavior because of the atomicity of the operation $fetch\&store$. We omit the detailed definition of the two transitions.

Definition of OTS $\mathcal{S}_{\text{corr}}$ of the correct mutual exclusion protocol: $\mathcal{S}_{\text{corr}} \triangleq \langle \mathcal{O}_{\text{corr}}, \mathcal{I}_{\text{corr}}, \mathcal{T}_{\text{corr}} \rangle$.

- $\mathcal{O}_{\text{corr}} \triangleq \{pc_{\text{corr}} : \Upsilon \times P \to L, locked_{\text{corr}} : \Upsilon \to \mathcal{B}\}$
- $\mathcal{I}_{\text{corr}} \triangleq \{\upsilon_0 | pc_{\text{corr}}(\upsilon, p) = rs_{\text{corr}} \wedge \neg locked_{\text{corr}}(\upsilon_0)\}$
- $\mathcal{T}_{\text{corr}} \triangleq \{enter_{\text{corr}} : \Upsilon \times P \to \Upsilon, exit_{\text{corr}} : \Upsilon \times P \to \Upsilon\}$

Having the two OTSs $\mathcal{S}_{\text{flawed}}$ and $\mathcal{S}_{\text{corr}}$, we can formalize the dynamic update based on Definition 3, since the update conforms to the incremental updating model.

Definition of OTS $\mathcal{S}_{\text{upd}}$ for the dynamic update: $\mathcal{S}_{\text{upd}} \triangleq \langle \mathcal{O}_{\text{upd}}, \mathcal{I}_{\text{upd}}, \mathcal{T}_{\text{upd}} \rangle$.

- $\mathcal{O}_{\text{upd}} = \mathcal{O}_{\text{flawed}} \uplus \mathcal{O}_{\text{corr}} \cup \mathcal{O}'$
- $\mathcal{I}_{\text{upd}} = \{\upsilon_0 | \upsilon_0 \in \mathcal{I}_{\text{flawed}}, \neg started(\upsilon_0), \neg updated(\upsilon_0, p), \neg updated'(\upsilon_0)\}$
- $\mathcal{T}_{\text{upd}} = \mathcal{T}_{\text{flawed}} \uplus \mathcal{T}_{\text{corr}} \cup \mathcal{T}'$

$\mathcal{O}'$ and $\mathcal{T}'$ are the sets of observers and transitions, which are the same as the ones in Definition 3. The set P in the arity of observers in $\mathcal{O}'$ and transitions $\mathcal{T}'$ in $\mathcal{S}_u$ is the same as the one in $\mathcal{S}_{\text{flawed}}$ and $\mathcal{S}_{\text{corr}}$, indicating a set of process's identifiers. Namely, the state of each process in the old system is considered as an indexed sub-state, and there is only one unindexed sub-state, i.e., the value of *locked*. The update of each process and the shared Boolean variable *locked* is specified by *update* and *update'*, respectively. Transition *start* formalizes the starting of the update. We explain how the three transitions are defined to specify the dynamic update.

In the example, we assume that the dynamic update can be started at any moment. Thus, there is no extra condition for the transition *start* except that the update has not been started. After the transition *start*, the system starts to be updated. Thus, the effective condition of *start* and the change of the value observed by *started* can be defined by the following two equations:

```
eq $c\text{-}start(\upsilon)$ = not $started(\upsilon)$  .
ceq $started(start(\upsilon))$ = $true$ if $c\text{-}start(\upsilon)$ .
```

After an update starts, a process must be updated if it is in the remainder section and not yet updated. Thus, we declare the following equation to define the effective condition of *update*:

```
eq $c\text{-}update(\upsilon, p)$ =
   $started(\upsilon)$ and not $updated(\upsilon, p)$ and $pc_{\text{flawed}}(\upsilon, p) = rs$  .
```

Once a process is updated, it goes into the remainder section of the correct protocol, and is marked as updated. The following two equations specify this updating.

ceq $pc_{\mathrm{corr}}(update(\upsilon, p), p')$ =
    (if $p = p'$ then rs_{corr} else $pc_{\mathrm{corr}}(\upsilon, p')$ fi) if $c\text{-}update(\upsilon, p)$.
ceq $updated(update(\upsilon, p), p')$ =
    (if $p = p'$ then $true$ else $updated(\upsilon, p')$ fi) if $c\text{-}update(\upsilon, p)$.

Transition $update'$ can be defined likewise. It specifies the updating of the unindexed sub-state, i.e., the value observed by $locked_{\mathrm{flawed}}$. We assume that the value can be updated at any state after the update has been started. After being updated, the value observed by $locked_{\mathrm{corr}}$ is initialized by the value observed by $locked_{\mathrm{flawed}}$. The following two equations specify the effective condition and the initialization of the value observed by $locked_{\mathrm{corr}}$ due to the updating.

eq $c\text{-}update'(\upsilon)$ = $started(\upsilon)$ and not $updated'(\upsilon)$.
ceq $locked_{\mathrm{corr}}(update'(\upsilon))$ = $locked_{\mathrm{flawed}}(\upsilon)$ if $c\text{-}update'(\upsilon)$.

5.3 Verification of the Dynamic Update

One basic property the system should enjoy after being updated is mutual exclusion. Another property is *deadlock freedom* in that the system after being updated should never reach to a deadlock state. A deadlock state is that the value of $locked_{\mathrm{corr}}$ is true but no process is at the critical section. Thus, no process can enter the critical section and the value observed by $locked_{\mathrm{corr}}$ is always true, leading to deadlock.

Verification by Theorem Proving. For the mutual exclusion property, it is equal to say that for any state υ which is reachable from an initial state in $\mathcal{I}_u$ and any two processes p_1 and p_2, if both p_1 and p_2 are at the critical section of the correct protocol after the system is updated, p_1 and p_2 must be the same one. We use a state predicate $\mu : \Upsilon \times P \times P \to \mathcal{B}$ to formalize it.

eq $\mu(\upsilon, p_1, p_2)$ =
    $(pc_{\mathrm{corr}}(\upsilon, p_1) = cs_{\mathrm{corr}}$ and $pc_{\mathrm{corr}}(\upsilon, p_2) = cs_{\mathrm{corr}}$ implies $p_1 = p_2)$.

We prove in CafeOBJ that $\mu(\upsilon, p_1, p_2)$ is true for any p_1, p_2 in P, and any υ which is reachable from initial states in $\mathcal{I}_{\mathrm{upd}}$. The proof is based on structural induction. Three lemmas are used in the proof. The lemmas are also proved in the same way in CafeOBJ. We omit the details of the proof since it is not the emphasis of the paper.

Verification by Searching. For the deadlock freedom property, we verify that a deadlock state can never be reached from any initial state in $\mathcal{I}_u$ by using *searching* in CafeOBJ. If there are a limited number of processes running in the system, it is feasible to search all possible states that are reachable from initial states, and check whether there is a deadlock state.

In the example, we assumed that there are only two processes in the system and denoted them by p_i and p_j, respectively. Let υ_0 be the corresponding initial state in the old system. We want to verify that from υ_0 there is no such a state where p_i and p_j are not in the critical section of the correct protocol but the value of *locked* in the correct protocol is true.

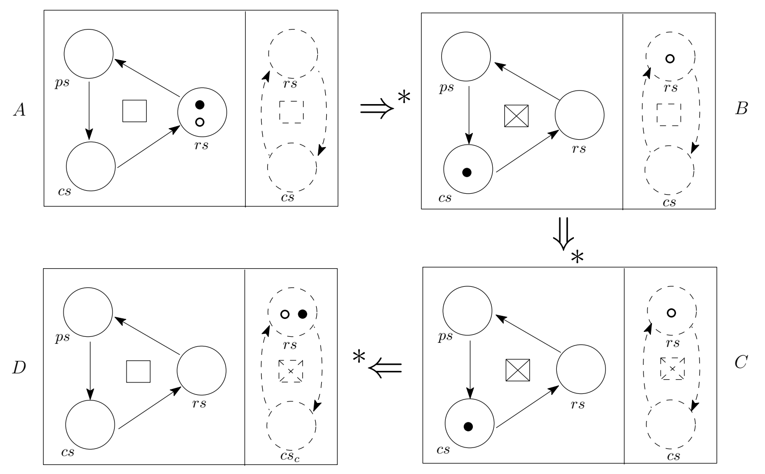
We define a state predicate $\rho : \Upsilon \rightarrow \mathcal{B}$ which returns true if a given state υ is a deadlocked state.

eq $\rho(\upsilon) = (pc_{\mathrm{corr}}(\upsilon, p_i) = rs$ and $pc_{\mathrm{corr}}(\upsilon, p_j) = rs$ and $locked_{\mathrm{corr}}(\upsilon))$.

We use the following CafeOBJ command to search a state υ from υ_0 such that $\rho(\upsilon)$ is true.

red υ_0 =(*,*)=>* υ suchThat $\rho(\upsilon)$.

CafeOBJ returns a state as shown by D in Fig. 1, which means that from the initial state υ_0 there is a path for the system to reach a deadlocked state.



The meaning of notations:
● : p_i, ○ : p_j, ◯ : location, □: *true*, ⊠ : *false*, $\Rightarrow^*$: n-step transition ($n \geq 0$).
Solid symbols:the flawed mutual exclusion protocol, dashed symbols: the correct protocol.

Fig. 1. A counterexample for the deadlock freedom property caused by the updating

The dynamic update is obviously not safe because the system after being updated may go into a deadlocked state, though the correct mutual exclusion protocol is deadlock free. From the counterexample shown in Fig. 1, we recognize the reason why deadlock happens. Diagram A in the figure denotes the initial state. It goes to B after p_i enters the critical section, and *locked* in the flawed protocol is true in B. Then, updating happens. Process p_j goes to the remainder section of the correct protocol, and *locked* in the correct protocol is set to be the value of *locked* in the flawed protocol, i.e., true, as shown in C. From C, p_i leaves the critical section in the flawed protocol, and gets updated at remainder section. The whole system reaches D, which is a deadlocked state. One solution is to set an updating condition for the update of *locked*. Namely, *locked* in the flawed mutual exclusion protocol can be updated only if it is false. We revised the OTS $\mathcal{S}_{\text{corr}}$ based on this change, and verified the deadlock freedom property again with the new OTS. CafeOBJ returns no solutions, meaning that the revised dynamic update does not cause deadlock after the system is updated.

6 Related Work

Several studies have been conducted on the correctness of dynamic software updating. They can be grouped into two classes. One studies the correctness at the code level, such as type safety [2,3] and version consistency [4] as described in Section 1. They are necessary properties that should be satisfied by any dynamic updates to make sure systems after being updated can run correctly. The other class is about the correctness at a higher abstract level, such as validity [16] and behavioral correctness [6]. They are also useful to analyze how a running system's behavior is changed due to updating. However, their formalization is still at the code level, which may bring difficulties for verification, e.g., the undecidability of validity due to the halting problem of programs [16]. In our earlier work, we classified dynamic updates into three classes according to how they are implemented, i.e., *interrupt model*, *invoke model*, and *relaxed consistency model* [7]. Regardless of the difference in implementation, updates in interrupt model and invoke model can be considered as instantaneous updates, and those in relaxed consistency model as incremental updates. In that sense, the classification of dynamic updates as instantaneous and incremental updates is more general.

A similar approach has been proposed to study the correctness of adaptive programs [27], i.e., the adaption of a system between programs due to the surrounding environment. They classified adaptions into different models such as one-point adaption, guided adaption and overlap adaption. They used a concrete example to illustrate their idea of how to formalize the adaptions. At the behavior level, the adaption of a system can be viewed as a special kind of "updating", and thus we believe that our formalization approach can also be used to formalize the adaption of systems.

7 Conclusion

We have presented an approach to formalizing and verifying the design of dynamic software updates in the OTS/CafeOBJ method. We classified dynamic updates into two models, i.e., *instantaneous updating model* and *incremental updating model*. A dynamic update that conforms to either of the two models can be formalized as an OTS, with which we can formally analyze the update by verifying whether it satisfies the desired properties. By verification we can understand better how a system's behavior is affected by an update, find updating points where an update can be safely applied, and detect some potential errors in an update such as errors in state transformation or updating condition. Our approach is general in that it is neither specific to a concrete update nor to the updates that are specific to a concrete programming language.

We are considering conducting more case studies on practical dynamic updates with the proposed approach. Potential cases are dynamic updates to some prevalent web applications such as *vsftpd* and *sshd*. Updates to these applications confirm to instantaneous updating model. Some are dynamic updates to multi-threaded applications such as *Apache* HTTP server. They have been used as benchmarks by some DSU systems such as POLUS [18], Ginseng [17]. However, they focus on the implementation of these updates, but pay little attention to their correctness such as whether the system after being updated satisfies the properties that are supposed to be satisfied by the system of the new version. Such questions can be answered by verifying these dynamic updates in our proposed approach.

References

1. Hicks, M., Nettles, S.: Dynamic software updating. ACM TOPLAS 27, 1049–1096 (2005)
2. Duggan, D.: Type-based hot swapping of running modules. In: Functional Programming, vol. 36, pp. 62–73. ACM (2001)
3. Stoyle, G., Hicks, M., Bierman, G., et al.: Mutatis mutandis: safe and predictable dynamic software updating. ACM TOPLAS 40, 183–194 (2005)
4. Neamtiu, I., Hicks, M., Foster, J., et al.: Contextual effects for version-consistent dynamic software updating and safe concurrent programming. In: POPL, vol. 43, pp. 37–49. ACM (2008)
5. Gupta, D., Jalote, P., Barua, G.: A formal framework for on-line software version change. IEEE Transactions on Software Engineering 22(2), 120–131 (1996)
6. 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)
7. Zhang, M., Ogata, K., Futatsugi, K.: Formalization and verification of behavioral correctness of dynamic software updates. Electr. Notes Theor. Comput. Sci. 294, 12–23 (2013)
8. Futatsugi, K., Goguen, J.A., Ogata, K.: Verifying design with proof scores. In: Meyer, B., Woodcock, J. (eds.) Verified Software. LNCS, vol. 4171, pp. 277–290. Springer, Heidelberg (2008)
9. Ogata, K., Futatsugi, K.: Proof scores in the OTS/CafeOBJ method. In: Najm, E., Nestmann, U., Stevens, P. (eds.) FMOODS 2003. LNCS, vol. 2884, pp. 170–184. Springer, Heidelberg (2003)
10. Ogata, K., Futatsugi, K.: Compositionally writing proof scores of invariants in the OTS/CafeOBJ method. J. UCS 19, 771–804 (2013)
11. Ogata, K., Futatsugi, K.: Simulation-based verification for invariant properties in the OTS/CafeOBJ method. Electr. Notes Theor. Comput. Sci. 201, 127–154 (2008)
12. Kong, W., Ogata, K., Futatsugi, K.: Towards reliable E-Government systems with the OTS/CafeOBJ method. IEICE Transactions 93-D, 974–984 (2010)
13. Hasebe, K., Okada, M.: Formal analysis of the *i*kp electronic payment protocols. In: Okada, M., Babu, C. S., Scedrov, A., Tokuda, H. (eds.) ISSS 2002. LNCS, vol. 2609, pp. 441–460. Springer, Heidelberg (2003)

14. Ogata, K., Futatsugi, K.: Formal verification of the horn-preneel micropayment protocol. In: Zuck, L.D., Attie, P.C., Cortesi, A., Mukhopadhyay, S. (eds.) VMCAI 2003. LNCS, vol. 2575, pp. 238–252. Springer, Heidelberg (2002)
15. Ogata, K., Futatsugi, K.: Formal analysis of the bakery protocol with consideration of nonatomic reads and writes. In: Liu, S., Araki, K. (eds.) ICFEM 2008. LNCS, vol. 5256, pp. 187–207. Springer, Heidelberg (2008)
16. Gupta, D., Jalote, P.: On-line software version change using state transfer between processes. Software: Practice and Experience 23, 949–964 (1993)
17. Neamtiu, I., Hicks, M.W., Stoyle, G., et al.: Practical dynamic software updating for c. In: PLDI, ACM SIGPLAN, pp. 72–83 (2006)
18. Chen, H., Yu, J., Hang, C., et al.: Dynamic software updating using a relaxed consistency model. IEEE Transactions on Software Engineering (99), 679–694 (2011)
19. Segal, M., Frieder, O.: On-the-fly program modification: Systems for dynamic updating. IEEE Software 10, 53–65 (1993)
20. Hayden, C.M., Smith, E.K., Denchev, M., Hicks, M., Foster, J.S.: Kitsune: Efficient, general-purpose dynamic software updating for c. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, pp. 249–264. ACM (2012)
21. Diaconescu, R., Futatsugi, K.: CafeOBJ report: The language. In: Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification, vol. 6 (1998)
22. Ogata, K., Futatsugi, K.: Some tips on writing proof scores in the OTS/CafeOBJ method. In: Futatsugi, K., Jouannaud, J.-P., Meseguer, J. (eds.) Goguen Festschrift. LNCS, vol. 4060, pp. 596–615. Springer, Heidelberg (2006)
23. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
24. Zhang, M., Ogata, K., Nakamura, M.: Translation of state machines from equational theories into rewrite theories with tool support. IEICE Transactions on Information and Systems 94-D, 976–988 (2011)
25. Nakamura, M., Kong, W., et al.: A specification translation from behavioral specifications to rewrite specifications. IEICE Transactions 91-D, 1492–1503 (2008)
26. Ogata, K., Nakano, M., Kong, W., Futatsugi, K.: Induction-guided falsification. In: Liu, Z., Kleinberg, R.D. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 114–131. Springer, Heidelberg (2006)
27. Zhang, J., Cheng, B.H.C.: Model-based development of dynamically adaptive software. In: ICSE, pp. 371–380. IEEE (2006)