An Algebraic Approach to Formal Analysis of Dynamic Software Updating Mechanisms

Min Zhang, Kazuhiro Ogata, Kokichi Futatsugi
Research Center for Software Verification & Graduate School of Information Science
Japan Advanced Institute of Science and Technology (JAIST)
Email: {zhangmin,ogata, futatsugi}@jaist.ac.jp

***Abstract*—Dynamic Software Updating (DSU) is a promising software maintenance technique, which aims at updating running software systems on the fly without incurring any downtime. The systems that require dynamic updating usually require high reliability assurance. Incorrect updating may cause them to behave erratically and/or even crash, and hence results in dreadful loss. However, there are few approaches to the study of the correctness of dynamic updating. In this paper, we systematically discuss the correctness of dynamic updating from a formal perspective, and present a first algebraic approach to formal analysis of it. The basic idea is to formalize dynamic updating systems as rewrite systems, with which we can analyze dynamic updates e.g. verifying their desired properties, or detecting incorrect update points, etc. The formal analysis helps us understand the behaviors of updated systems before we apply updates to the running systems, and hence improves the reliability of the the systems after being updated.**

***Index Terms*—Dynamic software updating, formal verification, correctness, algebraic formalization, rewriting logic**

I. Introduction

A. Background to DSU

Dynamic software updating (DSU) is a promising software maintenance technique of updating running software systems without incurring downtime. The systems that require dynamic updates are mainly those that provide 24x7 services such as web applications, traffic control systems, financial transaction system. The systems usually require a high reliability assurance. However, incorrect dynamic updates to them may cause systems to behave erratically and/or even crash, hence resulting in dreadful loss. Thus, correct dynamic updates to them are extremely important and must be guaranteed.

So far, a number of systems have been designed and implemented to apply dynamic updates to running software systems. Typical systems include Podus [1], OPUS [2], Ginseng [3], POLUS [4], and many others. These systems have been used in experiments for the dynamic updates to some typical server applications such as *vsftpd* (a commonly used FTP daemon), *sshd* (secure shell daemon) and some HTTP servers. However, it does not mean that updates applied by these systems to other applications are also correct. In other word, only testing is not enough to guarantee the correctness of an updating system. One reason is that dynamic updates depend upon not only the systems that are used to apply updates, but the systems to be updated and the points when updates are applied. Unless we are hundred percent sure of the correctness of a dynamic updating system, we cannot predict the next update to be applied is correct. The other reason is that we should not test it with the real running system which we are going to dynamically update.

B. Formal analysis of dynamic updating

Formal approaches, compared to testing, are more suitable for the analysis of the correctness of dynamic updating. A few studies have been conducted on the formalization and verification of the correctness of DSU systems. In each study, at least two aspects have to be discussed, i.e., the definition of correctness of dynamic updating and the approaches used for the formalization and verification.

Each study has a different definition of the correctness. Generally, they fall into two categories: 1) general ones that are independent from target systems, and 2) specific ones that are determined by target systems. Typical definitions of the former type include those proposed by Kramer et al. [5], and Gupta et al. [6]. In [5], an update is considered correct if two systems are *observationally equivalent*. Namely, updated program behaves exactly as the old program does. As observed by Bloom and Day [7], this view is rather restrictive. Gupta et al. proposed the notion of validity in [6]. An update is called valid if after update a system must eventually reach some state which is reachable by executing the new program from some of its initial states [6]. However, such definition is both too permissive and too restrictive as observed by Hayden et al. [8]. They, instead, proposed a flexible way of describing the correctness of dynamic updating by allowing programmers who are in charge of updating to describe the properties that their concrete dynamic updates should be satisfied with.

Another aspect of each study is about the approaches used to analyze the correctness. In [6], they showed the undecidability of valid updates, and proposed some sufficient conditions based on concrete program models. However, their approach is lack of tools to check if the conditions are satisfied by concrete programs. In [8], Hayden et al. proposed to verify user-defined properties by merging an existing program version with its update. The merged program is equivalent to the updated program. To achieve this, they must know potential update points beforehand. Thus, the approach is specific to their concrete updating system, and cannot be applied to those systems that support dynamic updates without explicit update points such as POLUS.

C. Our contributions

We systematically discuss the correctness of a dynamic update with both the two aspects, i.e., the common properties that are shared by all correct dynamic updates, and the specific properties that should be enjoyed by concrete updates. One basic common property is that dynamic update will never cause crash of running systems, which we call *no-crash* property. We give a formalization of it in a concrete updating mechanism based on a concrete program model. However, only the *no-crash* requirement does not suffice. Users are required to explicitly state the properties that guarantee that their target systems can behave correctly after updating. Such properties are specific to concrete target systems.

We also propose a general approach to formalizing and analyzing dynamic updating in an algebraic way. Dynamic updating systems are formalized as rewrite systems. A rewrite system takes a formalized program and updates to be applied to the program, and generates all possible update results. The update results can be used to verify against specified properties by existing rewriting-based reachability analysis approaches such as model checking.

To demonstrate the applicability of the approach, we formalize a dynamic updating approach which supports updates to multi-threaded applications, and show how to specify and verify desired properties of the correctness of dynamic updating. Like other existing approaches, the formalization is based on concrete program models, but can be easily extended to more complex ones.

In short, the contributions of this paper are as follows:

1) A systematic discuss on the correctness of dynamic software updating;
2) An algebraic approach to the formalization and verification of the correctness;
3) A prototype formalization of an updating mechanism that supports dynamic updates to multi-threaded approaches, based on a simple program model.

The rest of this paper is organized as follows. Section II presents an overview of DSU, and a concrete updating approach which we formalize in our example. Section III discusses the correctness of dynamic updating. Section IV briefly explains the logic foundation underlying our approach, i.e., rewriting logic, and a rewriting-based specification language i.e., CafeOBJ, which we use in our formalization. Section V describes the formalization approach, and the verifications based on the formalization. Section VI discusses some related work, and Section VII concludes the paper and mentions some future work.

II. Dynamic Software Updating

Dynamic software updating is opposite to 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, the former supports on-the-fly updates to running systems without shutting them down. It is particularly useful to update those systems that have to provide 24x7 services.

Several systems that support DSU have been developed so far. Some only support limited types updates that do not modify global states such as OPUS [2], and Ksplice [9], while some support flexible changes such as Ginseng [3], POLUS [4], etc. One main approach to dynamic update is to use patches that contain all the changes between the software of the old and new versions. Some apply patches at fixed update-points in the running software, while some support dynamic updates without update-points explicitly given. Some complete dynamic updates instantaneously, while some incrementally. We focus on those DSU systems that support flexible changes in our study since it is more preferable in practice for its flexibility. Particularly, we choose the updating mechanism underlying POLUS to formalize for its generalities i.e., support of incremental updates to multi-threaded applications, and needlessness of update-points.

A. Changes reflected in patches

Typically, flexible changes that are supported by DSU systems are the additions and modifications of types of global variables and functions. A global variable is considered *changed* if its type is changed and the identifiers are literally the same. A literally different variable is called *added* in the new version. A function is considered *changed* if it refers to changed or added global variables, but has a literally identical function in the old version with the same signature. A function is *added* if it is either literally different or of different signatures from other functions in the old version.

Such changes are reflected by *patches*. Deletions of global variables or functions are not necessarily reflected in patches, because they usually do not require special handling. Since the update is incrementally applied by POLUS, there is a period when there are both old and new code and data coexisting in running systems. To keep the consistent between the new and old data, data must be convertible from one version into the other. There are two corresponding functions in a patch used for the conversions, which are called *state synchronization functions* (or state mapping functions). Once a state is modified, corresponding synchronization function is invoked to convert the changed state into the one of the other version.

A patch also provides mapping information between old and new functions, between old and new global variables, and between functions and variables. Such information is used to assist update when the patch is applied.

B. Patch application

A patch is first compiled to a library. A dynamic update can be considered as a process of applying a pre-compiled patch to a running system incrementally until the system behaves as the new one. Functions are granularity of updates, namely that functions are treated as black boxes during updates. Updates to functions are achieved by *function indirection*, i.e., to insert an indirect *jump* instruction in the prologue of the original old function to force all the calls to the old function to call the corresponding new function.

ver-0

```
int r;
void area()
{
  sem_wait(&mutex)
  r = 2 * r;
  sem_post(&mutex)
}
```

ver-1

```
float r;
void area()
{
  sem_wait(&mutex)
  r = 2.5 * r;
  sem_post(&mutex)
}
```

Fig. 1. Two versions of a C program

A patch application consists of three steps, i.e., 1) patch initialization, 2) state synchronization, and 3) patch termination. In the first step, the running system to be patched is first suspended, and some initialization work is done such as loading a pre-compiled patch into memory, resolving symbols, and registering the patch information. Old functions are mapped to new functions by function indirection. All changed global variables are write protected, so that any modifications to these variables after the suspended system is resumed will cause the invokes of corresponding state synchronization functions. The second step is to resume the suspended system and do state synchronizations whenever any access to write protected variables occurs. In the last step, the update is terminated once the condition of termination is satisfied, and then some cleanup work is done. The termination condition is that when no threads are executing on the old changed global variables. A thread is executing on an old changed global variable when it is executing a function which refers to that variable. After the update is terminated, the running system behaves as it is executing the new program.

C. A dynamic update example

We explain the basic updating mechanism underlying POLUS with a simple example as shown in Fig. 1. Global variable r is considered changed from ver-0 to ver-1 because of the change of its type. For convenience, let r_0 and r_1 be the integer r and float r, and $area_0$, $area_1$ be the functions in ver-0 and ver-1, respectively. Suppose r_1=r_0 in the state synchronization function from r_0 to r_1, and r_0=(int)r_1 from r_1 to r_0. We assume that there are two threads T_0 and T_1 executing in $area_0$ and $area_1$ during updating. Before they occupy the mutex, we assume that r_0 is 3 and r_1 is 3.0. Due to the mutual exclusion, we assume T_1 finishes $area_1$ first. Then, r_1 becomes 7.5, and r_0 becomes 7 by executing r_0=(int)r_1. After T_0 finishes $area_0$, r_0 becomes 14. Meanwhile, r_1 becomes 14.0 by executing r_1=r_0. After T_0 finishes $area_0$, the update can be safely terminated because no threads refer to the old variable r_0.

III. Correctness of DSU systems

As mentioned in Section I, correctness of a DSU system is extremely important to its applicability. The definition of correctness is important to its study in formal methods. Without a clear definition of the correctness, it would be impossible to formalize DSU systems, not mentioning verifications.

However, there is not a standard definition of it. Gupta et al. have proposed to formalize target programs as abstract state machines, and defined the validity of dynamic updates at state machine level [6]. In their framework, the old and new programs are formalized as two state machines, and dynamic updates are defined as a process of transforming an old state s at some point into a new one by a state mapping function $\mathcal{S}$, and executing the new program from the new state s. Such an update is considered valid if by executing new program from s' it can eventually reach a new state s'' which is also reachable from some initial state of the new program. However, a valid update may be not a useful one. For instance, if we map any old states to some initial state of the new program, it is obvious that update at any point is valid according to the definition. However, such updates essentially can be considered as static updates. Another issue is that it is generally undecidable to checking if an update is valid due to the undecidability of halting problem. It would bring difficulties to the verifications of the validity of given updates.

It would be more reasonable to define correctness in a more flexible way, i.e. correctness is defined with respect to target systems. Programmers who are in charge of updating systems are required to describe the properties that any correct update must be satisfied with. This approach has been adopted by Hayden et al. [8].

In our approach, we consider both the two aspects of the correctness, namely that the properties that are shared by all correct dynamic updates, and those that are specific to concrete dynamic updates to be applied and target systems to be updated. All the properties are formalized as logic formulae. We can analyze or verify these formulae with formalized updates.

Finally, we discuss three common properties in this section. A straightforward one is that any correct updates must never cause target systems to crash during or after updating. Since crashes may be caused by bugs inside systems or patches rather than updates, these crashes should be assumed to never happen. Namely, we are only concerned with the crashes caused by update itself.

Another common property is called *type safety* [10], [11], which means functions of different versions can never refer to those data of inappropriate types. It is different from the notion of type safety in programming languages, where a language is called *type safe* if it can discourage or prevent type errors. By dynamic updating if an update is applied incorrectly, new program may access old data, and hence cause problems. For instance, function `f` is defined as `f(A a){...}` in an old program, where `A` is a user-defined type. Suppose that `A` is changed in the new version. After update, whenever `f` is called, the argument taken by `f` should be of the new type of `A`. Otherwise, problems may be caused. A correct update should guarantee type safety.

The last common property is called *version consistency* of function calls. Dynamic updates may cause unexpected function calls which may cause problems. For example, suppose that in an old version a function `f()` is defined to call `g()` and `h()` sequentially, e.g., `f(){g();h();}`. In the new version, new `f()` only calls `g()`, and `g()` calls `h()`, e.g., `f(){g();}` and `g(){h();}`. When an update takes place

in old `f()` before old `g()` is called, the call to old `g()` is then redirected to new `g()`. New `g()` then calls `h()`. After `g()` returns, old `f()` calls `h()` again. Such function calls will never happen either in old or new version, but may happen during updating, which may cause unexpected results. It is called *version inconsistency problem* [12], [4].

The above three properties are prerequisites for any correct dynamic updates, which are independent from concrete systems. However, they are not enough to guarantee an update is a desired one because even if there are no crashes after update, updated systems may not behave as expected. Additional properties are necessary to describe expected behaviors that will eventually happen or unexpected ones will never happen after updates. Such properties should be provided by programmers with concrete target systems. For instance, we may need to guarantee that r_1 is always greater than or equal to r_0 during any dynamic updates to the programs in Fig. 1 in some context.

IV. Rewriting Logic and CafeOBJ

A. Rewriting logic

Rewriting logic was proposed by Meseguer as a logical framework which generalizes both equational logic and term rewriting [13]. It is well suited to formalization of language semantics, particularly for concurrency. It is a logic to reason the change in a concurrent system. Each rewrite rule is a general pattern in the form of $l \Rightarrow r$ for a basic action that can occur in a system. l and r are patterns of states. A state of a fragment of a state that is matched by l can be rewritten by the rule. The matched fragment of the state is transformed into the corresponding instance of r.

Rewriting logic allows us to reason about other complex changes are possible in a system, based on the basic actions that are axiomatized by rewrite rules.

Theoretically, a rewrite theory is a triple $\mathcal{R} = (\Sigma, E \cup A, R)$, with $(\Sigma, E \cup A)$ an equational theory, where Σ is a signature and A is a set of axioms corresponding to equational attributes such as associativity, commutativity, etc., and R a set of labeled rewrite rules that are applied *modulo* the equations E and axioms A. Intuitively, the equational theory $(\Sigma, E \cup A)$ specifies the "statics" of a system, i.e., the algebraic structure of system states. The rewrite rules R specifies the "dynamics" of a system, i.e., the transitions between systems.

Rewriting logic has a wide range of applications, e.g., formalizations of the semantics of programming languages such as λ-calculus, (mini-) ML, the π-calculus [14], etc., and specifications and verification of software systems such as communication protocols and security protocols [15].

B. CafeOBJ

In our approach, we use CafeOBJ as the formalization language to specify dynamic updates and the properties. CafeOBJ is a successor of the famous algebraic specification OBJ with new features. For example, it supports rewriting logic specifications which are based on a simplified version of Meseguer's rewriting logic [16]. Many methods have been proposed to specify and verify software systems in CafeOBJ [17]. It has been successfully used to verify the design of software systems such as mutual exclusion protocols, communication protocols, and authentication protocols.

We take the formalization of natural numbers for example. We use N to denote the data type of all the natural numbers. Then, we declare an operator $0 :\rightarrow N$ to represent zero, and an operator $s : N \rightarrow N$, which is used to construct the successor of a given number. For instance, $s(n)$ denotes the successor of n. Next, we show how to define the meaning of operators by equations. For instance, we formalize the addition operation on natural numbers. An infix operator $+$ is declared i.e., $+ : N \times N \rightarrow N$. For any natural numbers x, y, the following two equations hold:

$$x + 0 = x \tag{1}$$

$$x + s(y) = s(x + y) \tag{2}$$

Basic units in CafeOBJ are called *modules*. In a module, there are declarations of *sorts* (which are essentially types of the data being defined), operations among them, equations for the meaning of operators, or rewrite rules. Each of them is optional in a module.

The following module specifies a set of natural numbers as the sort `Nat`.

```
module! NAT {
  [ Nat ]
  op 0  :      -> Nat
  op s_ : Nat -> Nat
}
```

Keyword `module!` declares a module, followed by a user-given name, e.g. `NAT`. A sort is declared between `[` and `]`, e.g. `Nat`. Operators are defined by the keyword `op`, followed by an operator name, arity and coarity of the operator. For example, `0` is an operator with `Nat` as its coarity and no arity. An operator without arity is also called a constant. Operator `s_` is an infix one with `Nat` as both of its arity and coarity. The underbar indicates where the argument of `s_` goes. Semantically, `0` is interpreted as the natural number 0, and `s_` denotes the successor of a given natural number.

Next, we define the addition operation on natural numbers as `_+_`, as shown in the following module.

```
module! NAT+ {
  extending(NAT)
  op _+_ : Nat Nat -> Nat {assoc comm}
  vars M N : Nat
  eq M + 0 = M .
  eq M + s N = s(M + N) .
}
```

Module `NAT+` extends `NAT` by the keyword `extending`. Operator `_+_` is associative and commutative, declared by `assoc` and `comm` which are called operator attributes. Two variables `M` and `N` of sort `Nat` are declared by `vars`. Equations are declared by `eq` (or `ceq` for conditional equations). The meaning of `_+_` is defined by the two equations in the module.

Operationally, equations are understood by considering them as rewrite rules from left to right, which is how CafeOBJ "computes". Namely, a term is reduced by applying possible equations until no more equations can be applied.

As shown in the above example, there is a clear correspondence between the equational formalizations, such as equations 1 and 2, and CafeOBJ specifications. For convenience, we use the formalizations directly in this paper without giving the corresponding CafeOBJ specifications.

CafeOBJ also supports rewrite rules, which are used to specify transitions of dynamic systems. For instance, in multi-threaded software systems the state of a thread is changed from ready to running once it occupies CPU. We use the rule i.e., $\langle T, \mathit{ready}, \ldots\rangle \Rightarrow \langle T, \mathit{running}, \ldots\rangle$ to specify the transition. We assume that each thread is represented as a tuple, T denotes the process id, and *ready* and *running* denote thread states i.e., ready and running, respectively. The change of other values of the thread are omitted from the rule by using "...".

$$
\begin{aligned}
\mathit{Program} &::= \mathit{DeclStmts}\ \texttt{call main()}\ \mathit{Funcs}\\
\mathit{DeclStmts} &::= \textit{empty} \mid \mathit{DeclStmts}\ \mathit{DeclStmt}\\
\mathit{DeclStmt} &::= \texttt{declare}\ \mathit{Vid}\\
&\mid\ \ \texttt{declare}\ \mathit{Vid} = \mathit{Exp}\\
\mathit{Funcs} &::= \mathit{MainFunc} \mid \mathit{MainFunc}\ \mathit{SubFuncs}\\
\mathit{MainFunc} &::= \texttt{main()}\{\mathit{Statements}\ \texttt{return}\}\\
\mathit{SubFuncs} &::= \textit{empty} \mid \mathit{SubFuncs}\ \mathit{SubFunc}\\
\mathit{SubFunc} &::= \texttt{func}\ \mathit{Fid}\texttt{()}\{\mathit{Statements}\ \texttt{return}\}\\
\mathit{Statements} &::= \textit{empty} \mid \mathit{Statements}\ \mathit{Statement}\\
\mathit{Statement} &::= \mathit{AssiStmt} \mid \mathit{CallStmt} \mid \mathit{Exp}\\
\mathit{AssiStmt} &::= \mathit{Var} := \mathit{Exp}\\
\mathit{CallStmt} &::= \texttt{call}\ \mathit{Fid}\texttt{()}\\
\mathit{Exp} &::= \mathit{Vid} \mid \mathit{Exp}\ \mathit{Ops}\ \mathit{Exp} \mid \mathbb{Z}\\
\mathit{Ops} &::= + \mid - \mid * \mid /
\end{aligned}
$$

Fig. 2. Syntax of a simple program model

V. Formalization and Verification

We show the formalization and verification of dynamic updating mechanisms in rewriting logic. As an example, we choose the updating mechanism of POLUS, and formalize it based on a simple but typical imperative program model.

A. Formalization

The formalization includes three parts, i.e., formalizations of target programs, patches, and the updating mechanism.

1) Formalization of programs: We first need to fix a program model and formalize its semantics. For simplicity, the first program model we consider is like any imperative language e.g., C, or Pascal, with only basic statements such as assignments and function calls, and basic numerical expressions such as addition, multiplicity. We also assume that all variable are integers, and functions do not take argument and return values for the same reason.

The syntax of our first program model is shown by Fig. 2. A program in this model consists of a list of declaration statements, followed by a statement to call `main` function, i.e., `call main()`, and a set of functions which at least include a `main` function. A declaration statement initializes an integer variable Vid by the value of an expression Exp, or 0 by default. The body of each function is a list of assignment statements and function calls, ended with `return`.

We formalize the syntax of the program model. First, we need to formalize basic elements in the model, e.g., integers, variables, expressions, statements, and so on. The formalization of integers is similar to natural numbers, as mentioned in Section IV-B. There is a built-in module in CafeOBJ formalizing integers $\mathbb{Z}$. Without the loss of generality, we only give the formalization of expressions in detail as demonstration. Formalizations of other elements are similar.

Let E be sort of expressions. According to the syntax, an integer is also an expression, which can be specified by letting $\mathbb{Z}$ be a subsort of E. We declare an infix operator "_+_: $E \times E \to E$" to formalize the addition "+" of two expressions.

Suppose that we have formalized sets of functions and lists of statements as sorts $Set\{\mathcal{F}\}$ and $List\{\mathcal{S}\}$, where $\mathcal{F}$ and $\mathcal{S}$ are sorts of functions and statements, respectively. We declare a sort $\mathcal{P}$ for the programs, and an operator $pg : Set\{\mathcal{F}\} \times List\{\mathcal{S}\} \to \mathcal{P}$ to construct a program e.g. $pg(S_f, L_s)$ with a set S_f of functions and a list L_s of declaration statements. Each program that conform to the model can be represented in this form.

Next we specify the semantics. Semantics of the simple program model is similar to any imperative language. It is formalized in a similar way as proposed by Goguen [18]. In Goguen's approach, program states are represented by configurations, which are in the form of $\langle P, K, M, s\rangle$, consisting of a program P, a call stack K, a memory store M, and statement s (or statements) to be executed. A call stack K records current functions that are being executed. M is a key-value memory store, recording the values of declared variables. Each statement is represented as a rewrite rule, specifying how configurations are changed by the statement.

The formalization of the semantics is partially shown in Fig. 3. We take the second rule for instance. It means that after the execution of a statement which declares a new variable V with the value of expression Exp, a new mapping $V \mapsto \mathit{Val}$ is added to the memory, where *Val* equals $\mathit{eval}(M, Exp)$. Function *eval* is used to evaluate the value of Exp with respect to the memory store M. The second last rule gives the meaning of function call. When a function is called, e.g., `call` F, F is pushed into stack K, and the statements in the body of function F is to be executed. Function *getStmts* returns the statements in the body of a given function F which is defined in P. The last rule formalizes the semantics of `return`, which denotes the function on the top of the stack returns.

2) Formalization of Patches: A patch consists of four parts, i.e., 1) a set of new functions, some of which are changed

$$
\begin{array}{lll}
\langle P, K, M, \texttt{declare } V\rangle & \Rightarrow \langle P, K, (V \mapsto 0)\ M, 0\rangle & \\
\langle P, K, M, \texttt{declare } V = Exp\rangle & \Rightarrow \langle P, K, (V \mapsto \mathit{Val})\ M, \mathit{Val}\rangle & \mathit{Val} = eval(M, Exp) \\
\langle P, K, M, Exp_1\ Ops\ Exp_2\rangle & \Rightarrow \langle P, K, M, \mathit{Val}\rangle & \mathit{Val} = eval(M, Exp_1\ Ops\ Exp_2) \\
\langle P, K, ((V \mapsto \mathit{Val})\ M), V\rangle & \Rightarrow \langle P, K, ((V \mapsto \mathit{Val})\ M), \mathit{Val}\rangle & \\
\langle P, K, ((V \mapsto \mathit{Val})\ M), V := Exp\rangle & \Rightarrow \langle P, K, ((V \mapsto \mathit{Val'})\ M), \mathit{Val'}\rangle & \mathit{Val'} = eval((V \mapsto \mathit{Val})\ M, Exp) \\
\langle P, K, M, \texttt{call } F\texttt{()}\rangle & \Rightarrow \langle P, push(K, F), M, \mathit{Stmts}\rangle & \mathit{Stmts} = getStmts(P, F) \\
\langle P, K, M, \texttt{return}\rangle & \Rightarrow \langle P, pop(K), M, 0\rangle &
\end{array}
$$

Fig. 3. Formalization of the semantics

from old functions, and some are newly added ones; 2) a set of mappings from new changed functions to old functions, 3) a set of mappings from functions to the set of variables which they refer to, and 4) a set of mappings from old variables to new variables. They are formalized by sorts $Set\{\mathcal{F}\}$, $Map\{\mathcal{F}, \mathcal{F}\}$, $Map\{\mathcal{F}, \mathcal{V}\}$ and $Map\{\mathcal{V}, \mathcal{V}\}$, respectively.

We take the definition of $Map\{\mathcal{V}, \mathcal{V}\}$ for example. A mapping from an old variable to a new one consists of four elements, i.e., the old variable, the new variable, two relations from one to the other. For instance, we assume V_1 and V_1' are the old and new variables, respectively, and they are always equal. We represent the mapping between them as $\{V_1, V_1', V_1 := V_1', V_1' := V_1\}$. Statement $V_1 := V_1'$ is used to synchronize from V_1' to V_1, and $V_1' := V_1$ from V_1 to V_1'.

We declare sort $\mathcal{H}$ for patches, and an operator $patch$: $Set\{\mathcal{F}\} \times Map\{\mathcal{F}, \mathcal{F}\} \times Map\{\mathcal{F}, \mathcal{V}\} \times Map\{\mathcal{V}, \mathcal{V}\} \rightarrow \mathcal{H}$, which constructs a patch with a set of functions and three sets of mappings respectively.

3) Formalization of updating mechanism: Next, we formalize the core part of a DSU system, i.e., its updating mechanism. Updating mechanisms may be different in different DSU systems. Some are based on interrupted model while some are invoke model. Interrupted model means a running program can be interrupted for updates at any moment during its execution. Invoke model means that a running program calls an update procedure which is pre-compiled at some fixed points in the program. Details on their differences can be referred to [19].

As an example, we formalize the updating mechanism of POLUS. The updating mechanism is called relaxed consistency model [4], which can be considered as an extension of the general interrupted model. The basic idea of the relaxed consistency model is that during updating both the data of old and new versions co-exists, and the codes run concurrently. This mechanism is well suited to the dynamic updates to multi-threaded applications.

Because POLUS supports dynamic update to multi-threaded applications, we extend the program model described in Section V-A1 by introducing thread into it. We introduce a new keyword `spawn` into the program model. It is followed by a function, e.g. `spawn` F`()`. A new thread is then created, beginning with executing the function that follows `spawn`. To formalize the semantics of `spawn`, we revise the definition of configurations, i.e., a configuration consists of a program P, a memory store M, and a process C, represented by $\langle P, M, C\rangle$.

In multi-threaded program model, a process C is formalized as a set of threads which executes the same program. Threads share the same program, but have their own call stacks and program counter to indicates next statements to be executed. Let $\mathcal{T}id$ and $\mathcal{K}$ be the sorts of thread identifiers and call stacks. We declare sort $\mathcal{T}$ for threads, and operator $thread : \mathcal{T}id \times \mathcal{K} \times List\{\mathcal{S}\} \rightarrow \mathcal{T}$ to construct a thread, e.g. $thread(T, K, L_s)$ with a thread identifier T, a call stack K, and a list L_s of statements to be executed. Let $Set\{\mathcal{H}\}$ be the sort of sets of threads, and $\mathcal{H}$ be a subsort of it. Threads in a process are concatenated by an associative and commutative infix operator "_|_".

The semantics of `spawn` is formalized by the following rewrite rule:

$$\langle P, M, (thread(T, K, ((\texttt{spawn } F\texttt{()})\ L_s) \mid C)\rangle \Rightarrow$$
$$\langle P, M, (thread(T, K, L_s) \mid thread(T', \emptyset_s, \texttt{call } F\texttt{()}) \mid C)\rangle$$

T, T' are thread identifiers, K is the call stack of thread T, $\emptyset_s$ denotes an empty call stack, and L_s denotes the statements to be executed by T.

Due to the change of configurations, we redefine the semantics of each statement in the new model. As an example, the semantics of assignment statements is formalized by the following rule:

$$\langle P, (V \mapsto \mathit{Val})\ M, (thread(T, K, ((V := Exp)\ L_s) \mid C)\rangle \Rightarrow$$
$$\langle P, (V \mapsto \mathit{Val'})\ M, (thread(T, K, L_s) \mid C)\rangle$$

Val' is the value of $eval((V \mapsto \mathit{Val})\ M, Exp)$. Note that this rule makes sense only under the assumption that all global variables are able to accessed by at most one thread at the same time. Such assumption is reasonable because mutual exclusion to accessing critical sections is a basic requirement to multi-threaded programming. It is worth mentioning that even though the thread in the rule appears to be the first one in the process, it is actually an arbitrary one due to the commutativity and associativity of "_|_". Semantics of other statements can be redefined similarly.

Finally, we formalize the process of patch injection in POLUS. It consists of three main steps to inject a patch into a running system, i.e., *initialization*, *state synchronization*, and *termination*. We define four update states, i.e., *bfUpdate*, *updating*, *done* and *abort* to denote four status of applying an update, i.e., before updating, being updated, update being finished, and update being aborted, respectively. We extend configurations with update states and patches to formalize the updating mechanism. Namely, a configuration is in the form of $\langle U, P, M, C, H\rangle$, consisting of an update state U , a program P, a memory store M, a process C (essentially, a set of

Initialization: $\langle \textit{bfUpdate}, P, M, C, H\rangle \Rightarrow \langle \textit{updating}, P, init(M, H), C, H\rangle$

Function indirection: $\langle U, P, (V \mapsto \textit{Val})\ M, (thread(T, K, ((\texttt{call}\ F\ \texttt{()})\ L_s)) \mid C, H)\rangle \Rightarrow \begin{cases} \langle U, P, M, (thread(T, push(K, F), L'_s\ L_s) \mid C), H\rangle & \text{if } U = \textit{bfUpdate} \\ \langle U, P, M, (thread(T, push(K, F'), L''_s\ L_s) \mid C), H\rangle & \text{otherwise} \end{cases}$

where, $F' = getFunc(F, H)$, $L'_s = getStmts(P, F)$ and $L''_s = getStmts(P, F, H)$

State synchronization: $\langle U, P, (V \mapsto \textit{Val})\ M, (thread(T, K, ((V\ :=\ Exp)\ L_s)) \mid C, H)\rangle \Rightarrow \begin{cases} \langle U, P, (V \mapsto \textit{Val'})\ M, (thread(T, K, L_s) \mid C), H\rangle & \text{if } U = \textit{updating} \\ \langle U, P, (V \mapsto \textit{Val'})\ M', (thread(T, K, L_s) \mid C), H\rangle & \text{otherwise} \end{cases}$

where, $\textit{Val'} = eval(M, Exp)$ and $M' = sync(M, V, \textit{Val'}, H)$

...

Termination: $\langle \textit{updating}, P, M, C, H\rangle \Rightarrow \langle \textit{done}, P, M, C, H\rangle$ if $term?(C, H)$

Abort: $\langle \textit{updating}, P, M, C, H\rangle \Rightarrow \langle \textit{abort}, P, M, C, H\rangle$ if $exception?$

Fig. 4. Formalization of the updating mechanism of POLUS

threads), and a patch H. The main definition is shown by Fig. 4.

The first rule specifies the initialization step at the beginning of applying a patch H to the running process P. The update state is changed from *bfUpdate* to *updating*, and the memory store M is changed into $init(M, H)$, which represents the memory store after initialization. Function $init$ takes a memory store M and a patch H, and returns a new memory store where new variables are declared and initialized with the corresponding old variables by the state synchronization function given in H.

The second rule specifies function indirection. After the initialization of patches, when a function F is called, the call will be redirected to its corresponding new function F' (if there is). Function *getFunc* returns the new function of F from H, and *getStmts'* returns the statements in F'.

The third rule formalizes state synchronizations during updating. State synchronization takes place when the values of variables are modified. For instance, after an assignment e.g., $V\ :=\ Exp$ is executed by some thread, the new value *Val'* is assigned to V, and meanwhile, its corresponding old or new value is updated according to *Val'* and the state synchronization function in H. Given M, V, *Val'*, and H, function $sync$ returns the new memory store M' after synchronization.

The fourth rule formalizes the termination of updates. Predicate *term?* takes the running process C and the patch H, and checks whether the current update can be safely terminated. For each changed old variable V in H, *term?* checks each thread in C if there are functions that refer to V being executed. If that is the case, update cannot be terminated. If for each variable there is no function that refers to it being executed in each thread, *term?* returns true.

The last rule formalizes the abort of updating due to occurrence of exceptions. Exceptions are denoted by a Boolean value *exception?*. Exceptions in our program model includes calling a non-existing function, reference to a variable that is not in current memory, division by zero, calling a non-existing function, etc. Handling exceptions and meaningless expressions in an algebraic setting reduces to handling subsorts and partial operations [20]. We take the formalization of division by zero for instance. We have defined a function $eval$ which takes a memory store M and an expression Exp as arguments, and returns the value of Exp. It is possible that division by zero occurs in Exp. To handle the exception, we declare a "supersort" of $\mathbb{Z}$, and denote it by $\mathbb{Z}'$. We define a constant exc_0 of $\mathbb{Z}'$ to represent the division by zero exception. When it occurs in an expression Exp, $eval$ returns exc_0 as the value of Exp. It is defined by the following equation.

$$eval((V \mapsto 0)\ M, V'\ /\ V) = exc_0 \tag{3}$$

The first argument says that in the memory store the value of variable V is 0, and the second argument says V is a divisor in the expression.

B. Verification

We specify the formalization above in CafeOBJ, and obtain an executable specification, with which we can verify the properties of specified updating mechanism.

1) Initial configuration: For verification, we need first to provide initial configurations to which the rewrite rules in Section V-A3 can be applied. As mentioned previously, a configuration consists of an update state U, a program P, a memory store M, a process C that executes P, and a patch H to be applied to C. We formalize P and H as described in Section V-A1 and V-A2 respectively. They will never change during executing and updating. Initially, M is empty before P is executed. The stack of each thread in C is empty, and the statement that each thread is about to execute is to call `main` function. The update state in initial configurations is *bfUpdate*. For instance, given a program P and a patch H, the initial configuration can be represented as $\langle \textit{bfUpdate}, P, \emptyset_m, thread(T_1, \emptyset_s, \texttt{call main()}), H\rangle$, where, $\emptyset_m$ denotes an empty memory store.

2) Formalization of properties: Next, we need to determine and formalize the properties to be verified. As discussed in Section III, some common properties must be guaranteed, such as no-crash, type safety, and version consistency.

The formalization of no-crash property is straightforward. Namely, there are no such configurations where the update state is *abort* and they can be reached from initial configurations. We define a predicate *isAbort?* that takes a configuration

c and returns true if the update state of c is *abort*, and false, otherwise. It is represented by the following equation:

$$isAbort?(\langle U, P, M, C, H\rangle) = \begin{cases} true & \text{if } U = abort \\ false & \text{otherwise} \end{cases}$$

Let $\mathcal{R}_\mathcal{C}$ be the set of all reachable configurations. Informally, a configuration is called *reachable* from an initial configuration c_0 if c can be eventually reached from c_0 by iteratively applying the rewrite rules. We formalize no-crash of updates as that for each c in $\mathcal{R}_\mathcal{C}$, *isAbort?*(c) is false.

As for type safety, because in our programming model function do not take any arguments, and there is only one type, i.e., integer, of variables, type unsafety cannot occur. We only give the idea of formalizing type unsafety in more complex models where functions can take arguments. The type unsafety can be formalized by a kind of exceptions, as we specify the division by zero exception in Section V-A3. Let us consider the example of type unsafety in Section III. After being updated, whenever `f` is called, we check whether the argument of `f` in the memory store is of the new type of `A`. If the answer is no, we say a type unsafety exception is caused. Then, we define a predicate e.g., *typeSafe?*, which returns true if there are exceptions caused by type unsafety in configurations. Then, type safety can be formalized as an invariant property, e.g., for each c in $\mathcal{R}_\mathcal{C}$, $\neg$*typeSafe?*(c).

Next, we consider version consistency property. Instead of formalizing the property itself, we formalize one sufficient condition of it, i.e., old functions that are changed in the new version should never call new functions [4]. For instance, old `f()` should never call new `g()` in the example in Section III. To achieve this, the mapping from old `g()` to new `g()` should be deleted. That is, new `g()` is considered as an added one, and hence no function indirection is applied to old `g()`, which is also the solution in POLUS [4]. The formalization of the sufficient condition is straightforward. We define a predicate $verCon?$ which returns true if no old changed functions are calling new functions in a configuration. Whenever a new function is called, we check the function on the top of the stack. If the function is an old one, $verCon?$ returns true, and false otherwise. It can be defined by the following equation:

$$verCon?(\langle U, P, M, (thread(T, K, (\texttt{call } F\,\texttt{()}\,L_s))|C), H\rangle) = \begin{cases} false & \text{if } U \neq \textit{bfUpdate} \wedge isOC(top(K)) \wedge isNew(F') \\ true & \text{otherwise} \end{cases}$$

where, $F' = getFunc(F, H)$, and predicate $isOC(F)$ returns true for a function F if F is an old function and it is changed in the new version. The sufficient condition can be formalized as that for each c in $\mathcal{R}_\mathcal{C}$, $verCon?(c)$ is true.

For user-defined properties, their formalization depends on concrete programs and patches. We take the user-defined property in III for example. The property is formalized as that for each c in $\mathcal{R}_\mathcal{C}$, $geq(c)$ is true, where predicate geq returns true if $r_1 \geq r_0$ in a configuration c.

3) Verification: With the formalizations of both DSU systems and properties, we can verify specified properties by existing approaches, such as *searching* and *model checking*. Searching is mainly used to verify invariant properties such as the properties in above section. Model checking also supports liveness properties.

We consider the verification of version consistency property as an example. Let P and H be the formalizations of the program and the corresponding patch in the example in III. We verify if no version inconsistency will occur by applying H to a process running P. We assume the initial configuration is the same as the one in Section V-B1, and denote it by c_0. In CafeOBJ, we use the following command to automatically find counterexamples of the property.

`red` c_0 `=(`m`,`n`)=>*` c `suchThat` $\neg verCon?(c)$.

Command `red` in CafeOBJ is used for the reduction of terms based on equations and rewrite rules. Parameters m and n are natural numbers, limiting the number of final configurations to be found and the number of transitions to be applied. A Boolean term that follows `suchThat` denotes the condition that the configurations to be found should be satisfied with. CafeOBJ searches all the configurations that are resulted from any possible updates. CafeOBJ returns a counterexample by finding a configuration where old `f()` calls new `g()`. If we delete the mapping from old `g()` to new `g()` from P, no counterexamples are found, which coincides with the solution to version inconsistency problem of POLUS. That is, new `g()` is considered as a *newly added* instead of a *changed* function of old `g()`. We found that, more generally, a function f in new version is considered as *changed* of a function in old version if it is satisfied with not only the conditions in Section II, but the one that all the functions in old version that call f must be not *changed* functions. The strengthened conditions guarantee the version consistency of any updates by POLUS.

Verifications of no-crash and type safety properties are similar. As for user-defined properties, we can verify them by searching if they are invariant properties. For others such as liveness properties, we can verify by model checking [1].

Similarly, given a program and a patch that is based on our program model, we formalize them and then define an initial configuration with them. Then, we can verify with the formalized updating mechanism the properties of dynamic updates by applying the patch to the program.

VI. Related Work

A. Related approaches

There are several approaches to formal analysis or verification of the correctness of DSU systems, which inspire our approach. Gupta et al. study the "validity" of dynamic updates with concrete program models [6], which has become a well-accepted way to study the correctness of DSU systems. Duggan proposed the notion of type safety of dynamic updates [11], and Neamtiu et al. proposed a formal approach to version

[1]Currently, CafeOBJ does not provide model checking facilities for the verification of liveness properties. However, a sibling language called Maude [21] supports model checking. Thus, we can choose Maude as our specification language in the proposed formalization approach, which is one piece of our future work.

consistency [12]. These are considered in our approach as basic properties that should be enjoyed by any DSU systems. Hayden et al. pointed out another aspect of the correctness of dynamic updates, i.e., properties specific to target systems, and the approach to verifications by program merging. We believe that both the two aspects of correctness are equally important, and need to be verified. For instance, type safety and version consistency are necessary to guarantee systems after being updated can behave as expected. They, however, do not suffice to guarantee that unexpected behave must never occur in an updated system. Their approaches give significant impact on our definition of the correctness, and the approach to the formalizations and verifications of DSU systems, e.g., formalization based on concrete program models. A main difference is that our approach is general but not specific to concrete DSU systems and properties, in that it is extensible to formalize the dynamic updates of different programming models. With the formalization, we can verify both the two aspects of the correctness with the off-the-shelf tools such as model checkers.

B. Related algebraic specification systems

We choose CafeOBJ as the specification language and verification system. However, our approach is not only restricted to using CafeOBJ. Besides, there are also other algebraic specification languages and verification systems, such as Maude [21], ELAN [22]. We believe that our approach is also applicable by using them as specification languages in order to use their powerful verification facilities. For instance, Maude also provides powerful model checking facilities such as Maude LTL model checker [23], LTLR model checker [24], besides *searching* functionality. They are useful to verify not only invariant properties, but liveness properties such as version consistency defined in Section V-B2.

C. Formal semantics of programming languages

In our approach, formalization of the semantics of program models plays an important role in the formalization of DSU systems. Unlike the formalization of DSU systems, formal semantics of program models have been widely studied. The $\mathbb{K}$ framework is proposed by Roşu et al. based on rewriting logic as a framework of formal semantics [25]. Semantics of languages such as C have been specified in $\mathbb{K}$ [26]. We believe that our approach can also be extended by using $\mathbb{K}$ to specify the semantics of more complicated programming models. Then, we can formalize DSU systems that are based on these model, and verify the correctness of those dynamic updates supported by the systems. For instance, there are many DSU systems that are developed for the dynamic updates based on C-like programming models, such as Ginseng [3], and POLUS [4]. We can extend our approach by using the specification of the semantics of C in $\mathbb{K}$ to represent the dynamic updates for C programs. It would be one piece of our future work.

VII. Conclusion and Future Work

We have proposed a rewriting-based approach to formalizing and analyzing the correctness of DSU systems. We systematically discussed the meaning of the correctness of dynamic updates from formal perspective, and proposed to define the correctness from two aspects, i.e., common properties shared by all dynamic updates, and the properties that are specific to concrete target systems to be updated. We formalized a non-trivial updating mechanism underlying a powerful live dynamic updating system POLUS as an example, and specify it by an algebraic specification language CafeOBJ. We also showed the verifications of desired properties with the formalized dynamic updating mechanism. By formal verification, we can find desired updates-points for correct updating, gain deep insight into the behaviors of a formalized DSU system, and hence improve the reliability of applying dynamic updates to running software systems. To the best of our knowledge, this is the first approach that is based on rewriting logic to the formalization and verification of the correctness of DSU systems. We believe our approach can also be supported by other powerful rewriting-based specification and verification tools, and can be extended to formalize dynamic updates with more complex program models.

As we have mentioned in Section III, currently there is still not a standard answer to the question that "what is a correct dynamic update?". The essence of dynamic updates from the formal perspective is still unclear. From our current achievements, there are two possible directions for future progress in this study. The first is to study the correctness at a higher abstract level to verify it at the behavior level, e.g., whether systems after being updated behavior as expected, and whether unexpected behavior must never occur. In that direction, we can model a program as an abstract state machine, and verify the behavior level property by using theorem proving or model checking. The other direction is to focus on the verification of code-level properties by extending the proposed approach to concrete DSU systems for more complex programming model. For instance, based on the formal semantics of C in $\mathbb{K}$ framework, we could formalize the updating mechanism of POLUS for formal analysis of dynamic updates to general C programs, which is one piece of our future work.

Acknowledgement

This work was funded in parts by Japanese Grants-in-aid for Scientific Research with project number: 23220002. The authors would like to thank Parallel Processing Institute (PPI) of Fudan University for their kind host of the first author's three-month visiting to study POLUS. The authors also would like to thank those anonymous reviewers who have gave many useful comments to the paper.

References

[1] O. Frieder and M. Segal, "On dynamically updating a computer program: From concept to prototype," *Journal of Systems and Software*, vol. 14, no. 2, pp. 111–128, 1991.

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

[3] I. Neamtiu, M. W. Hicks, G. Stoyle, and M. Oriol, "Practical dynamic software updating for c," in *PLDI*. ACM SIGPLAN, 2006, pp. 72–83.

[4] H. Chen, J. Yu, C. Hang, B. Zang, and P. Yew, "Dynamic software updating using a relaxed consistency model," *IEEE Transactions on Software Engineering*, no. 99, pp. 679–694, 2011.

[5] J. Kramer and J. Magee, "The evolving philosophers problem: Dynamic change management," *IEEE Transactions on Software Engineering*, vol. 16, no. 11, pp. 1293–1306, 1990.

[6] D. Gupta, P. Jalote, and G. Barua, "A formal framework for on-line software version change," *IEEE Transactions on Software Engineering*, vol. 22, no. 2, pp. 120–131, 1996.

[7] T. Bloom and M. Day, "Reconfiguration and module replacement in argus: theory and practice," *Software Engineering Journal*, vol. 8, no. 2, pp. 102–108, 1993.

[8] C. M. Hayden, S. Magill, M. Hicks, N. Foster, and J. S. Foster, "Specifying and verifying the correctness of dynamic software updates," in *4th VSTTE, LNCS 7151*. Springer, 2012, pp. 278–293.

[9] J. Arnold and M. Kaashoek, "Ksplice: Automatic rebootless kernel updates," in *EuroSys*. ACM, 2009, pp. 187–198.

[10] G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu, "Mutatis mutandis: safe and predictable dynamic software updating," *ACM Tranc. Programming Languages and Systems*, vol. 40, no. 1, pp. 183–194, 2005.

[11] D. Duggan, "Type-based hot swapping of running modules," in *Functional Programming*, vol. 36, no. 10. ACM, 2001, pp. 62–73.

[12] I. Neamtiu, M. Hicks, J. Foster, and P. Pratikakis, "Contextual effects for version-consistent dynamic software updating and safe concurrent programming," in *POPL*, vol. 43, no. 1. ACM, 2008, pp. 37–49.

[13] J. Meseguer, "Rewriting logic as a semantic framework for concurrency: a progress report," in *7th CONCUR, LNCS 1119*. Springer, 1996, pp. 331–372.

[14] P. Thati, K. Sen, and N. Martí-Oliet, "An executable specification of asynchronous pi-calculus semantics and may testing in maude 2.0," *ENTCS*, vol. 71, pp. 261–281, 2004.

[15] J. Meseguer, "Software specification and verification in rewriting logic," *Nato Science Series III: Comput. and Sys.*, vol. 191, pp. 133–194, 2003.

[16] R. Diaconescu and K. Futatsugi, "Logical foundations of cafeobj," *TCS*, vol. 285, no. 2, pp. 289–318, 2002.

[17] K. Futatsugi, "Formal methods in cafeobj," in *6th FLOPS, LNCS 2441*. Springer, 2002, pp. 1–20.

[18] J. Goguen and G. Malcolm, *Algebraic semantics of imperative programs*. The MIT Press, 1996.

[19] M. Hicks and S. Nettles, "Dynamic software updating," *ACM Transactions on Programming Languages and Systems*, vol. 27, no. 6, pp. 1049–1096, 2005.

[20] J. Goguen and J. Meseguer, "Order-sorted algebra i: Equational deduction for multiple inheritance, overloading, exceptions and partial operations," *Theoretical Computer Science*, vol. 105, no. 2, pp. 217–273, 1992.

[21] M. Clavel, F. Durán, and et al., "All about Maude," *LNCS 4350, Springer*, 2007.

[22] P. Borovanskỳ, C. Kirchner, H. Kirchner, P. Moreau, and C. Ringeissen, "An overview of elan," *2nd WRLA, ENTCS*, vol. 15, pp. 55–70, 1998.

[23] S. Eker, J. Meseguer, and A. Sridharanarayanan, "The Maude LTL model checker," in *4th WRLA, ENTCS 71*. Elsevier, 2002, pp. 162–187.

[24] K. Bae and J. Meseguer, "State/event-based LTL model checking under parametric generalized fairness," in *23rd CAV, LNCS 6860*. Springer, 2011, pp. 132–148.

[25] G. Roşu and T. F. Şerbănuţă, "An overview of the K semantic framework," *Journal of Logic and Algebraic Programming*, vol. 79, no. 6, pp. 397–434, 2010.

[26] C. Ellison and G. Roşu, "An Executable Formal Semantics of C with Applications," in *39th POPL*. ACM, 2012, pp. 533–544.