An Event-based Formal Framework for Dynamic Software Update

Shengwei An    Xiaoxing Ma    Chun Cao    Ping Yu    Chang Xu
State Key Laboratory for Novel Software Technology at Nanjing University
Institute of Computer Software, Department of Computer Science and Technology, Nanjing University
Nanjing 210023, Jiangsu, China
njuasw@gmail.com, {xxm, caochun, yuping, changxu}@nju.edu.cn

Abstract—Dynamic Software Update (DSU) is a technique to upgrade running programs without shutting them down. DSU can improve system availability and maintenance flexibility. However, its adoption in practice is still limited due to the risk of system misbehavior that careless DSU may bring. To reduce this risk we propose a formal framework for the specification and verification of DSU. Different from previous approaches where DSU is described from the viewpoint of program's internal state transitions, our framework focuses on program's external behavior and its effect on its environment. This more abstract view avoids over specification of DSU and allows for better DSU flexibility. Based on this framework, we also devise a mechanism that automatically synthesizes runtime monitors to improve DSU timeliness without compromising its safety.

Keywords—dynamic software update; formal methods;

I. INTRODUCTION

Through the lifecycle of software systems, developers may constantly release updates to fix bugs, enhance functionality, or change their behavior as requirements evolve. Normally, these updates are applied to a production system through three steps: shutting down the system, installing new-version program and restarting the system. However, this kind of offline updates are unacceptable in some circumstances where non-stopping service are mandatory, such as financial transaction processing and transport controlling. Besides, the disruptions caused by frequent offline updates can be annoying for everyday software applications.

Dynamic software update (DSU) is a technique to upgrade a running program to a new version on the fly. Operationally, DSU traps the running program at some state, transforms the state to a proper state of the new program, and then executes the new program starting from this transformed state [1], [2], [3]. DSU preserves valuable program state and minimizes disruption during an update [4], which can help in ensuring high service availability. DSU also makes software upgrades transparent to users and thus provides more flexibility for maintenance. In the past decades, many DSU systems have been developed in the research community [2], [4], [5], [6].

However, DSU has been rarely used in real-world practice, mainly because there is a risk that DSU may cause misbehaviors that would never present in either the old or the new version of the program [3], [7]. It is far from trivial to eliminate these misbehaviors. The reason is twofold. First, programs are not typically developed with DSU in mind. Second, the state of the running program, and also the state of its environment, are not predictable when DSU is triggered.

To manage the risk, various formal models have been proposed for the specification and verification of DSU [3], [8], [9], [10]. However, some of them, such as [3], are too loose to ensure the safety of DSU. Others, such as [8], [10], tend to cause over-specification, because they take an operational view of DSU, and one must explicitly specify what kind of cross-version program traces are allowed.

In this paper we try to use a more abstract view that focuses on programs’ interactions with their environments rather than their internal state transitions. Let’s use a two-person game metaphor [11] to explain the intuition. A program is playing a game against the environment. In the middle of the game the program is replaced with a new version. For the new program to win the game, the important thing is not the current state of the old program but the future behavior of the environment. The latter is affected by previous interactions between the old program and the environment.

In our framework, programs, as well as their environments, are modeled with Labeled Transition Systems (LTSs) [12]. System requirements are specified in Fluent Linear Temporal Logic (FLTL) [13]. DSU is allowed if the new version program will correctly continue with the environment and keep satisfying the requirements. In this way, we lift the abstraction level from program states to interaction events, and constrain DSU with system requirements and environment models, which is more natural and easier than explicitly considering cross-version program state transitions.

Another common problem of previous formal DSU models is that they are too conservative. Using static analysis, they only allow updates to happen at program states that are always safe. This can be harmful to the timeliness of DSU. With our framework it is possible to reduce this conservativeness by just requiring the update to be safe for the current status of the environment. We exploit runtime monitors to get current environment status. Runtime monitors are automatically synthesized from program and environment models and system requirements. Monitors can be deployed on-demand when DSU request arrives, and thus no overhead exists during normal execution.

The intended contribution of this paper are twofold. First, we propose a formal framework to define DSU and its correct-
ness using LTS and FLTL. The framework helps avoid over-
specification of DSU and allows for better DSU flexibility, 
thanks to its event-based specification and explicit environment 
modeling. Second, we derive a runtime monitoring mechanism 
to improve the timeliness of DSU without compromising its 
safety.

The remainder of this paper is organized as follows. Section II gives some preliminaries about LTS and FLTL. We discuss our understanding of DSU and the motivation for a new framework in section III, and then present the framework formally in section IV. Section V details the runtime monitor-
based approach to timely DSU. We overview related work in 
Section VI before concluding the paper in Section VII.

II. PRELIMINARIES

Our framework uses LTS [12] to model programs and their 
environments, and use FLTL [13] to specify requirements. We 
briefly introduce LTS and FLTL, following the notations used 
in [14] and [15].

A. Labelled Transition System

LTS is a state transition system where transitions are 
labelled with actions. The set of actions (also called events) of 
an LTS is called its communicating alphabet by which the 
modeled system interacts with the environment.

Definition 1. (Labelled Transition System) Let States be 
a universal set of states, Act be the universal set of action 
labels. A Labelled Transition System (LTS) is a tuple $E = (S_E, A_E, \Delta_E, s_0)$, where $S_E \subseteq \text{States}$ is a finite set of states, $A_E \subseteq \text{Act}$ is a finite alphabet, $\Delta_E \subseteq (S_E \times A_E \times S_E)$ is a transition relation, and $s_0 \in S_E$ is the initial state.

Definition 2. (Parallel Composition) Given LTSs $M = (S_M, A_M, \Delta_M, s_{M0})$ and $E = (S_E, A_E, \Delta_E, s_{E0})$, parallel composition $||$ is a symmetric operator such that $E||M$ is the LTS $E||M = (S_{EM}, A_{EM} \cup A_M, \Delta, (s_{E0}, s_{M0}))$, where $\Delta$ is the smallest relation that satisfies the rules below, where $\ell \in A_{EM}$:

1) if $\ell \in A_E \setminus A_M$ and $(s, \ell, s') \in \Delta_E$, then $((s, t), \ell, (s', t')) \in \Delta$;
2) if $\ell \in A_E \setminus A_M$ and $(t, \ell, t') \in \Delta_M$, then $((s, t), \ell, (s', t')) \in \Delta$;
3) if $\ell \in A_M \cap A_E$ and $(s, \ell, s') \in \Delta_E$ and $(t, \ell, t') \in \Delta_M$, then $((s, t), \ell, (s', t')) \in \Delta$.

Definition 3. (Trace) Consider an LTS $E = (S, A, \Delta, s_0)$. A sequence $\pi = \ell_0, \ell_1, \ldots$ is a trace in $E$ if there exists a sequence $s_0, \ell_0, s_1, \ell_1, \ldots$, where for every $i \in \mathbb{N}$ we have $(s_i, \ell_i, s_{i+1}) \in \Delta$.

In this paper we restrict attention to LTSs $E$ that $\forall s \in S_E, s$ is reachable.

B. Fluent Linear Temporal Logic

In event-based models, states are characterised by the 
behavior that originates in these states rather than state variables [13]. FLTL is a method of describing abstract states of 
LTS which is based on events [14].

Definition 4. (Fluent) A fluent is a triple $FL = (I_{FL}, T_{FL}, \text{Init}_{FL})$, where $I_{FL}, T_{FL} \in \text{Act}$ is respectively the set of initiating and terminating actions, and $\text{Init}_{FL}$ is the initial boolean value of $FL$. Obviously, $I_{FL} \cap T_{FL} = \emptyset$.

Each action $\ell \in \text{Act}$ induces a fluent denoted by $\ell = (I_{\ell}, \text{Act} \setminus \{\ell\} \mapsto \text{false})$. It means that when $\ell$ occurs, this fluent $\ell$ becomes true, and becomes false when any other action occurs.

The trace $\pi = \ell_0, \ell_1, \ldots$ satisfies a fluent $FL$ at position $i$, denoted $\pi, i \models FL$, iff. at least one of the following holds:

- $\text{Init}_{FL} \land (\forall j \in \mathbb{N} : 0 \leq j \leq i \implies \ell_i \not\in T_{FL})$
- $\exists j \in \mathbb{N} : (j \leq i \land \ell_j \in I_{FL}) \land (\forall k \in \mathbb{N} : j < k \leq i \implies \ell_k \not\in T_{FL})$

Given a set of fluents $\mathcal{F}$ over $\text{Act}$, a FLTL formula is defined inductively using the standard Boolean operators, and the temporal operators $X$ (next) and $U$ (strong until) as follows:

- each fluent of $\mathcal{F}$ is a formula,
- if $\phi$ and $\psi$ are formulas, then so are $\neg \phi \lor \psi, X \phi, \phi U \psi$.

Boolean operators $\rightarrow$ is defined as: $\phi \rightarrow \psi \equiv \neg \phi \lor \psi$, and $\land$ as $\phi \land \psi \equiv (\neg \phi \lor \neg \psi)$.

Given an infinite trace $\pi$, the satisfaction of a formal $\phi$ at position $i$, denoted $\pi, i \models \phi$, is defined as follows [15]:

$\pi, i \models \neg \phi$ iff. $\neg (\pi, i \models \phi)$

$\pi, i \models \phi \lor \psi$ iff. $(\pi, i \models \phi) \lor (\pi, i \models \psi)$

$\pi, i \models X \phi$ iff. $\pi, i + 1 \models \phi$

$\pi, i \models \phi U \psi$ iff. $\exists j \geq i : \pi, j \models \psi \land \forall k < j : \pi, k \models \phi$

We also use standard abbreviations “true $\equiv \phi \lor \neg \phi$” and “false $\equiv \neg \phi$” and temporal operators $\Box$ (always), $\Diamond$ (eventually), W (weak until) as syntactic sugar, where $\Box \phi \equiv \true U \phi, \Diamond \phi \equiv \neg \Box \neg \phi$, and $\Diamond W \psi \equiv ((\Diamond U \psi) \lor \Box \phi)$.

We say $\phi$ holds in $\pi$, denoted $\pi \models \phi$, if $\pi, 0 \models \phi$. A formula $\phi$ holds in an LTS $E$ (denoted $E \models \phi$) if it holds on every infinite trace produced by $E$.

III. DYNAMIC SOFTWARE UPDATE

This section discusses our understanding of DSU and gives a running example.

A. Our Understanding

Despite the existence of many practices and research on 
dynamic software update, there is a lack of clear common 
understanding of how to do it correctly. Updating a program 
at runtime is not difficult per se because essentially codes 
are stored in the same way as data are under von Neumann 
arquitectures. However, what we actually care about is how to 
ensure that the system under updating behaves well before, 
during, and after the update. To clearly understand this well-
behavleness, or in other words the correctness of DSU, one 
must take a view from the interactions between the program 
and the environment around it.

Let us consider a two-person game metaphor proposed by 
Pnueli [11]. The environment and the program are like two 
players playing a game. Each player can only choose one of its 
actions in its turn. In the turn of the environment, the program

Authorized licensed use limited to: Purdue University. Downloaded on January 19,2022 at 03:36:21 UTC from IEEE Xplore. Restrictions apply.
can only observe how the environment behaves. The winning condition for the program is that the requirements are always satisfied whatever the environment does.

Suppose in the middle of the game, a new program player takes the place of the current program and continues the game. For the new player to win the game, he does not have to know the strategy and current state of the old program, but must be able to deal with the future behavior of the environment. However, the old program and its environment have interacted for a while before the substitution, and this affects the future behavior of the environment.

Most existing DSU approaches focus on how to map the current state of the old program to a proper state of the new program. However, with the above metaphor, we can see that this is too implementation-oriented and inflexible. Instead, we believe, first, analysis of DSU should be based on the external behavior of programs rather than their internal states, and second, explicit behavior models of environments, in addition to those of programs, are needed.

Based on whether the environment/requirement changes, there are four basic DSU scenarios.

1) Neither the environment nor the requirement changes. We want to update the program for reasons such as the enhancement of security or functionality, but we do not change requirements. So the winning condition does not change. In this case, the game is played by the environment and the old version of the program before the DSU. After the DSU, the new version of the program continues to play with the environment and wins.

2) The environment does not change but the requirements change. We want the program to do more things and we specify these changes by requirements. This case is like the first case except that the requirements change.

3) The environment changes but the requirements do not. The environment changes mainly for two reasons: a) we have the partial model of the environment at first, and then we understand it better and notice some unknown events or actions; b) the environment actually changes because the program works in a different situation. At the beginning of the game, the old program plays with the old environment. At some point, the environment changes, so the program needs to be updated to keep satisfying the requirements.

4) Both the environment and the requirements change. This case is more complex, as both the opponent (the environment) and the winning condition (the requirements) are different. We can regard this case as the combination of the second case and the third case.

When discussing the change of the environment, we only consider the case in which the new environment is an extension of the old one, i.e., the first kind shown in the third case. In this case, the new environment can execute the traces of the old environment.

B. Running Example

Figure 1 shows our running example. A Client gets a token from the Auth and uses it to call the Server. On receiving the call, the Server uses the Auth to verify the token and returns the data which it reads from the database. The Client then displays the data. Figure 2(a)–2(c) displays the models of the three components. Auth is functional and has only one immutable state. The actual concurrent execution of these three components is represented by their parallel composition in Figure 3.

At the beginning, this system is designed without the consideration of DSU. However, we now want to upgrade the Auth component to enhance security. The new version
uses a stronger encryption/decryption algorithm incompatible with the old version. Thus the new Auth would not verify the token issued by the old version. The environment and the program are relative to our focus, so we regard Auth in Figure 2(d) as the program and the parallel composition of the Client and the Server as the environment. The server and the client need to use the same version of Auth, otherwise they can not work successfully. In other words, the token obtained by the getToken[i] can only pass through the corresponding verify[i] action where i stands for the version. If the client calls the server with a token supplied by getToken[1] of Auth1 then the server must use verify[1]. The actual environment is shown in Figure 4.

Assume we replace Auth1 with Auth2 after the client uses getToken[1] but before the server uses the verify[1]. When the server needs verify[1] but the Auth2 has no such action, then deadlock happens. For simplicity, we assume that neither the environment nor the requirement changes. As we discussed above, we can only do DSU correctly when the system is outside a pair of getToken[i] and verify[i].

In this case the disadvantage of the “state mapping” approaches becomes evident. In these approaches DSU are allowed at quiescent states [8], [16] where updates are known to be safe. Because each of Auth1 and Auth2 has only one state, we can have exact one mapping between the two states in this case. However we can see from above discussion that the mapping is not guaranteed to be safe. So actually there is no quiescent state here according to Zhang’s definition in [8].

In fact when the environment is at state 0/3/4/5, we can update the program. We only have an optimistic DSU scheme rather than the pessimistic one here. Given an optimistic DSU scheme, the program itself can not decide whether it’s lucky enough to choose the correct DSU. We can use a monitor to monitor the recent history of the program and ensure only correct DSU traces occur. In this case, the monitor mainly observes whether the program is outside a pair of getToken and verify. When we want to do DSU, we start the monitor and if it says yes then the program can do a correct DSU. The details of the optimistic DSU algorithm is in Section V.

IV. A FORMAL FRAMEWORK FOR DSU

In this section, we formally define DSU and its correctness with LTS and FLTL. We also illustrate our formalization with the running example introduced in Section III.

A. Formalizing DSU

We use LTS to model programs and their environments, and FLTL to specify the requirements.

Definition 5. (System, Program, Environment, and Requirements) A system consists of a program and an environment. A program is modeled with a LTS P. An environment model is a LTS E. System requirements are specified by a set R of FLTL formulae.

We use subscripts to denote the version of programs, environments and requirements, and group them together as a tuple \((P_i, E_i, R_i)\) to denote the system of version \(i\). For simplicity, we use \((P_1, E_1, R_1)\) and \((P_2, E_2, R_2)\) to denote the system before and after DSU as default.

As we don’t want the collaboration of the environment and the program to reach a deadlock, we implicitly include the DEADLOCK_FREE property in the requirements as default. Furthermore, following [17], we restrict requirements to be in one of the following two forms: (1) global absence \(\Box \neg \phi\) and global universality \(\Box \phi\); (2) global response \(\Box (\phi \rightarrow \Box \psi)\), where neither \(\phi\) nor \(\psi\) has temporal operators. The restriction comes from the controller synthesis algorithm [15] on which our monitor generation algorithm is based. These two forms can express most requirements in practice. Matthew shows response, universality and absence patterns are the top three patterns and global scope is the most popular scope [17].

Let’s consider the running example introduced in Section III. As we mentioned before, the environment and the program is relative to our focus. We want to add some new functions into the client, so we regard the client as the program, whose first version \(P_1 = Cli_1\) is shown in Figure 2(a), and the parallel composition of the server and the Auth as the environment whose first version \(E_1\) is in Figure 5. For brevity, we demonstrate our formalization in the first scenario where \(E_1 = E_2\) and \(R_1 = R_2\).

The requirements \(R_1\) specify that it always holds that the client should eventually display the data if it calls the server to read data:

\[
\{ \text{DISPLAY}_\text{AFTER}_\text{CALL} = \Box (\text{call} \rightarrow \Diamond \text{display}) \}
\]

We want to log when the client requests the server for data, so Figure 6 shows the second version \(P_2\).

In this paper we assume that each individual version is correct, because we focus on the DSU, and the DSU can’t
be correct if the system itself is not correct. The correctness of a version means that the version satisfies the requirements. Given a set $R$ as requirements and a LTS $S$, we use $S \models R$ to denote $S \models \bigwedge_{\phi \in R} \phi$. In other words, when $R$ occurs on the right side of $\models$, $R$ stands for $\bigwedge_{\phi \in R} \phi$.

**Definition 6.**
(Correct version) A version $(P_i, E_i, R_i)$ is correct iff. $P_i \models E_i \models R_i$.

In our above example, we can easily verify that $P_1 \models E_1 \models R_1$ and $P_2 \models E_2 \models R_2$.

**Definition 7.**
(DSU scheme) A DSU Scheme is a set $DS = \{ (s, t) \mid s \in S_P \land t \in S_P \}$, $(s, t) \in DS$ means that by a $dsu$ action the old state $s$ jumps onto the new state $t$.

Note that our DSU scheme is defined on the abstract states in LTS. We use “$dsu$” to denote the DSU action. If the actions of the program and the environment include the keyword “$dsu$”, we can use the relabelling of LTS to rename those actions. Therefore, we assume that the actions of the program and the environment do not include the “$dsu$” for simplicity.

**Definition 8.**
(DSU LTS) Given a DSU scheme $DS$, a DSU LTS $D = (S_D, A_D, \Delta_D, s_D)\models S_P \cup S_P, A_D = \{ dsu \} \cup A_P, \Delta_D = \Delta_P \cup \Delta_P, s_D = s_P$. $S_D$ is the LTS generated by applying $\parallel$ to $P_1$ and $P_2$, where $S_D = \{ (s, dsu, t) \mid (s, t) \in DS \}$ and $s_D = s_P$.

Figure 7 shows a DSU LTS for $P_1$ and $P_2$. We will explain why it’s wrong after we give the definition of a correct DSU LTS and a correct DSU scheme. Because we assume that each version is correct, the behavior before the $dsu$ will not violate any old requirements and we only need to consider the correctness after the $dsu$.

We can use FTL to specify the period after a DSU as follows:

$$\text{AFTER\_DSU} = \{ \{ \text{dsu} \}, \varphi \land \text{false} \}$$

**Definition 9.**
(Correct DSU LTS, Correct DSU scheme) A DSU LTS $D$ is correct iff. $D \models E_2 \equiv \square(\text{AFTER\_DSU} \rightarrow R_2)$. A DSU scheme is correct iff. it can produce a correct DSU LTS.

As we point out in Section III that this paper only addresses the case where $E_2$ is an extension of $E_1$ and can interpret the traces in $E_1$, so we can make the parallel composition of $E_2$ and $D$.

The parallel composition of DSU LTS and $E_2$ is in Figure 8. The thicker purple edges indicate a trace which causes deadlock: $get\_Token, dus, log \# DEADLOCK\_FREE$, so Figure 7 gives a wrong DSU LTS. Figure 9 gives a correct DSU LTS.

This example tells us that we must take the old and new version of the program, the environment and the requirements into consideration to do a correct DSU.

Each execution of the program and the environment corresponds to a trace of the parallel composed LTS, so a particular DSU transition happens in a particular trace when we use LTS to model system. From now on, we treat the concept of a DSU transition and a DSU trace as the same.

We now define the DSU trace that is an execution of the system during which a DSU happens.
Definition 10. (DSU trace) Given \((P_1, E_1, R_1), (P_2, E_2, R_2)\) and a DSU scheme, a DSU trace is a trace \(\pi = \ell_0, \ell_1, \ldots, \ell_i, \ldots\) in \(D \parallel E_2\), which has exact one \(i\) such that \(\ell_i\) is dsu.

Intuitively, before the “dsu”, the trace is executed as it is on the \(P_1 \parallel E_1\), and on the \(P_2 \parallel E_2\) after the “dsu”. In other words, the \(i\)-length prefix of \(\pi\) is a prefix of a valid trace of \(P_1 \parallel E_1\) and \(\ell_{i+1}, \ldots\) is a postfix of a valid trace of \(P_2 \parallel E_2\). A DSU trace \(\pi\) is correct iff. \(\pi \models \Box (\text{AFTER_DSU} \rightarrow R_2)\).

Given a correct DSU scheme, we can produce a correct DSU LTS, all possible DSU traces constrained by this DSU LTS are correct. When the program is competing against the environment, it must execute on exact one of all possible traces, then the program can be updated dynamically and correctly. This kind of correct DSU scheme is a pessimistic one. An optimistic one contains correct DSU traces and also may contains incorrect DSU traces.

B. Case Study

Section III shows that there are four DSU scenarios. The first scenario is explained in Subsection IV-A. Here we demonstrate the rest scenarios.

1) Unchanged Environment and Changed Requirements:
In Subsection IV-A, we update the client to add a log action, but we do not specify this function in requirements \(R_2\). Now we add a property into \(R_2\):

\[
\text{LOGGED} = (\{\text{log}\}, \{\text{call}\}, \text{false})
\]

If we still use the DSU LTS in Figure 9, we will have some traces violate the \(\Box (\text{AFTER_DSU} \rightarrow R_2)\) property whose prefix is getToken, call, dsu.

This case shows again that the correctness of a DSU scheme is related to the requirements. A correct DSU LTS is in Figure 10.

2) Changed Environment and Unchanged Requirements:
The db action may take too long, and the server shown in Figure 11 may die of timeout and not reply to the client. If the client stills waits for the server to reply then the deadlock occurs as shown in Figure 12. We need to update the client to the new version in Figure 13. Figure 14 gives the environment \(E_2 = \text{Ser}_2 \parallel \text{Auth}_1\). The requirements are \(R_1 = R_2 = \{\text{DISPLAY}_\text{AFTER_CALL}\}\). A correct DSU is shown in Figure 15.

3) Changed Environment and Changed Requirements: It’s easy to demonstrate this case by the combination of the second scenario in section IV-B1 and third one in section IV-B2. We
want to add the log function and specify it by requirements, while we actually realize that the server may emit timeout without sending db data to the client. We omit the new version of client here, as we can easily understand it in the DSU LTS. Figure 16 gives the correct DSU LTS.

V. OPTIMISTIC DSU SCHEME AND MONITOR

We now detail the example introduced in Section III. We consider the first scenario, that is $E_1 = E_2 = Clnt_1 || Ser_1$ in Figure 4, $P_1 = Auth_1$ in Figure 2(d) and $R_1 = R_2 = \{\text{Display}_\text{AFTER}_\text{CALL}\}$. The only possible DSU scheme of the Auth is in Figure 17 by our definition of DSU scheme. Figure 18 gives the LTS $D || E_2$, dsu, $\text{getToken}[2]$, ... is a correct DSU trace but $\text{getToken}[1]$, $\text{dsu}$, ... is not. Therefore, we have an optimistic DSU scheme instead of a pessimistic one. Although optimistic schemes do not guarantee correct DSU, we can leverage a runtime monitor to ensure only correct DSU traces actually occur. The outline of our DSU algorithm is shown in Figure 19.

The problem of an optimistic DSU scheme is that, at runtime, a program knows exactly its own state but not its environment’s state, while the safety of DSU depends on both. For example, the old program always stays in state 0 in Figure 2(d) and correct DSU can start only if the environment is in state 0, 3, 4 or 5 in Figure 4. Probing environment state with some extra device would be too heavy and costly, if possible at all. With our model a light-weighted approach is possible. We attach a monitor to a system, which logs a finite recent history of the program’s interactions with environment. At runtime the monitor decides the safety of DSU based on the state of the system and the recent history. Essentially the state of the environment is (partially) inferred with the history. For example, assume we have a monitor that only informs the program to do DSU when the current state of the old program is state 0 and the recent history is $\text{verify}[1]$. We can see from Figure 4 that the current state of the environment must be one of state 0/3/4/5 after one $\text{verify}[1]$ and before one $\text{getToken}[1]$. Hence it ensures only correct DSU should occur.

With the benefits of our formalization, we can define the monitor clearly. An ideal and theoretical monitor is a controller $Ctrl$ in a LTS control problem [15]. The controller has only one controllable action $\text{dsu}$ and it can observe other actions $\text{Act}\setminus\{\text{dsu}\}$. For simplicity, we assume $E_1 = E_2$ and $R_1 = R_2$. It’s like a two-player game between the $Ctrl$ and $D || E_2$. The controller can only observe what the opponent does and restricts the occurrence of $\text{dsu}$ action. By the definition of LTS control problem, the parallel composition $Ctrl || D || E_2$ satisfies our requirements.

We use the tool MTSA [18] to synthesize a controller for the above example. Figure 20 shows the controller $Ctrl$.

By an ideal monitor, we can ensure the DSU should be correct by our definition, but it need to start monitoring from the initial state together with the program and the environment. We also need to know all the inner actions of the environment. Such a monitor is not practical, as the program can hardly know all of the inner states and actions of the environment. A desirable monitor only records some recent actions of the program and can start at any time.

We can induce a practical monitor from an ideal one (controller). The intuition is that given a recent history $rh$ of $P_1$, if the ideal monitor can ensure whatever the environment does, it can reach a state which can initiate $\text{dsu}$ action and then
our practical monitor allows the program to do DSU when it knows the recent history is $rh$. We use the notation $\pi[P_1]$ to denote the subsequence of a sequence $\pi$ and $\pi[P_1]$ only consists of $A_{P_1}$ and $S^* = \{s|\exists t \in S_{Ctrl} \cdot (s \xrightarrow{dsu} t)\}$ to denote the set of states that can start correct DSU. Given a recent history $rh$ of $P_1$, if for arbitrary prefix $\pi$ in the ideal monitor $Ctrl$, that $\pi[P_1]$ ends with $rh$ implies that execution of $\pi$ in $Ctrl$ goes into some state $s \in S^*$, then the practical monitor approves of the DSU request of the program. It is easy to see that a practical monitor may have false negatives but no false positives. That means we may miss some correct DSU traces. However, we never include any wrong DSU traces.

The practical monitor can be de facto implemented as a Deterministic Finite-state Automaton $DFA_M = (Q_M, \Sigma_M, \delta_M, s_M, F_M)$ induced from the controller. Each state of $DFA_M$ is a subset of $S_{Ctrl}$. We can obtain $DFA_M$ by Algorithm 1. Algorithm 1 is based on breadth-first search.

At first we don’t know what the current state of the controller is. It is denoted by the initial state of the DFA. Then we take each possible action of the old program, and select its closure to cover all possible actions of the environment until we can’t find new state to add into $Q_M$. Each accept state is a subset of $S^*$.

Figure 21 shows the practical monitor model obtained from

---

**Algorithm 1 Monitor Generation Algorithm**

**Input:**
The environment $E_2$  
The DSU LTS $D$  
The old system $P_1$

**Output:**  
Monitor $DFA_M = (Q_M, \Sigma_M, \delta_M, s_M, F_M)$

1: $Q_M = \Sigma_M = \delta_M = s_M = F_M = \{\varnothing\}$  
2: Synthesize the controller $Ctrl$.
3: if there is no $Ctrl$ then
4: \quad return $(Q_M, \Sigma_M, \delta_M, s_M, F_M)$
5: end if
6: \quad $S^* = \{s|\exists t \in S_{Ctrl} \cdot (s \xrightarrow{dsu} t)\}$
7: $s_M = S_{Ctrl}$
8: $Q_M.add(s_M)$
9: $\Sigma_M = A_{P_1}$
10: queue = new queue
11: queue.enqueue($s_M$)
12: while queue is not empty do
13: \quad $S = queue.dequeue()$
14: \quad for each act $\in A_{S_1}$ do
15: \quad \quad $S' = \{t|\exists s \in S \cdot (s \xrightarrow{act} t)\}$
16: \quad \quad $S' = S' \cup \{t|\exists s \in S' \exists a \in A_{E_2} \cdot A_{D} \cdot (s \xrightarrow{a} t)\}$
17: \quad \quad if $S' \notin Q_M$ then
18: \quad \quad \quad queue.enqueue($S'$)
19: \quad \quad \quad $Q_M.add(S')$
20: \quad \quad \quad if $S' \subseteq S^*$ then
21: \quad \quad \quad \quad $F_M.add(S')$
22: \quad \quad \quad end if
23: \quad \quad end if
24: \quad $\delta_M.add((S, act, S'))$
25: \quad end for
26: end while
27: return $(Q_M, \Sigma_M, \delta_M, s_M, F_M)$

---
Ctrl in Figure 20. The double rectangle denotes the accepted state of $DFA_M$.

**Proposition 11.** $DFA_M$ generated by Algorithm 1 only allows correct DSU traces to occur.

**Proof:** We just need to prove that given a recent history $rh$, if $DFA_M$ accepts $rh$, then for arbitrary prefix $\pi$ in $Ctrl$, that $\pi[P_1]$ ends with $rh$ implies that execution of $\pi$ in $Ctrl$ goes into some state $s \in S^*$. This is guaranteed by the closure operation in line 16 of Algorithm 1 and the definition of the accept states.

Therefore, $DFA_M$ can ensure that only correct DSU traces occur. Our algorithm in Figure 19 shows each time when we check the next state of the system, the $DFA_M$ runs the history input. But in practice, as the next history input is the extension of the current input, we can let the $DFA_M$ run with the system and just read the current action $a$.

Note that although this runtime monitor-based approach is safe, it does not guarantee that a DSU will eventually be allowed. This situation can happen when there are concurrent executions of the same component. In these cases one can use techniques similar to those proposed in [7] to achieve a DSU-able state.

VI. RELATED WORK

Formal treatment of dynamic adaptation of software systems has attracted a lot of research. However, most of them, such as [19], [20], [21], [22], [23], mainly focus on architectural changes. A survey of them can be found in [24]. Here we overview several formal models explicitly addressing behavioral changes in dynamic software update.

An early formal framework for dynamic software update was from Gupta et al. [3]. In this work, DSU is regarded as valid if it can lead to a reachable state of the new program in a finite amount of time. The authors proved that the validity of DSU is not decidable. However, obviously, the validity of DSU is insufficient for the correctness of DSU because system requirement is not taken into consideration.

Zhang and Cheng formalized dynamic software adaptations with finite state machines and LTL specifications [8]. They focused on a systematic development method for adaptive programs that can start from one program, undergo adaptation, and reach a second program. In their approach there is no explicit environment model. Program adaptation behavior is constrained by global specifications. So developers have to encode both application requirements and environment behavior into global specification. This can be tedious and error-prone for non-trivial environments. Their state machine-based formalism is also inconvenient for modeling interactive behavior of programs. Zhao et al.’s mLTL [10] is similar to this approach, and also suffers from these problems.

In a seminar paper, Kramer and Magee studied how to manage dynamic software changes [16]. They defined the concept of quiescence and proposed a safe method for dynamic reconfiguration of distributed systems. This method is general but very conservative. Later proposals, including ours and those discussed in the above paragraph, reduce the conservativeness with the information provided by program models, environment models and requirement specifications.

In a recent work Hayden et al. proposed a method for automatic verification of dynamic updates of C programs [25]. This work verifies dynamic updates against client-oriented specifications that constrain program’s external visible behavior.

While most approaches address one-step updating, Biyani et al. focus on the system’s behavior during overlap adaptations and specify them with adaptation lattices [9]. They proposed a framework to support the implementation of overlap adaptation.

Dynamical update also appears in network protocols. Woj-
cziechowski et al. defined a formal model for dynamic protocol update and proposed two algorithms to realize safe protocol update [26].

Our monitor synthesis is based on the controller synthesis technique for LTS [15]. Controller synthesis was studied by Pnueli and others [11]. The two-person game metaphor is also from them. However, our monitor synthesis is a very specific case of controller synthesis where the only controllable action dsu. It’s part of our future work to improve our algorithm by exploiting this specificity.

VII. CONCLUSIONS

In this paper, we formalize DSU with LTS models and FLTL specifications. Our formal framework explicitly models the external behavior of a program under updating, as well as its environment. It ensures that the new program correctly handles the environment left by the old program and continuously satisfying application requirements. Based on this framework, we also propose to automatically synthesize runtime monitors to improve DSU timeliness without compromising its safety. This runtime monitoring approach also exemplifies the flexibility and expressiveness of our framework.

Several issues require further investigations in our future work. In order to improve performance, we plan to generate runtime monitors directly, rather than transforming the problem to controller synthesis and using MTSA to do the job. We also need to experiment our approach on more realistic cases.

ACKNOWLEDGMENT

This work was supported by the National Basic Research 973 Program (Grant No. 2015CB352202), and National Natural Science Foundation (Grant Nos. 6142177, 91318301, 61321491) of China.

REFERENCES


