2015 IEEE International Conference on Software Quality, Reliability and Security

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-

978-1-4673-7989-2/15 $31.00 © 2015 IEEE
DOI 10.1109/QRS.2015.33

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 overviews 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 $S_E \subseteq States$ is a finite set of states, $A_E \subseteq Act$ 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 $M = (S_M, A_M, \Delta_M, s_{M_0})$ and $E = (S_E, A_E, \Delta_E, s_{E_0})$, parallel compostion $\|$ is a symmetric operator such that $E\|M$ is the LTS $E\|M = (S_E \times E_M, A_E \cup A_M, \Delta, (s_{E_0}, s_{M_0}))$, where Δ is the smallest relation that satisfies the rules below, where $\ell \in A_E \cup A_M$:

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 $\ell \in A_M \setminus A_E$ 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 $\pi = \ell_0, \ell_1, \cdots$ is a trace in E if there exists a sequence $s_0, \ell_0, s_1, \ell_1, \cdots$, 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 $FL = \langle I_{FL}, T_{FL}, Init_{FL} \rangle$, where $I_{FL}, T_{FL} \in Act$ is respectively the set of initiating and terminating actions, and $Init_{FL}$ is the initial boolean value of FL. Obviously, $I_{FL} \cap T_{FL} = \varnothing$.

Each action $\ell \in Act$ induces a fluent denoted by $\dot{\ell} = \langle \ell, Act \backslash \{\ell\}, false \rangle$. It means that when ℓ occurs, this fluent $\dot{\ell}$ 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 $\pi, i \vDash FL$, iff. at least one of the following holds:

- $Init_{FL} \wedge (\forall j \in \mathbb{N} \cdot 0 \leqslant j \leqslant i \rightarrow \ell_i \notin T_{FL})$
- $\exists j \in \mathbb{N} \cdot (j \leqslant i \wedge \ell_j \in I_{FL}) \wedge (\forall k \in \mathbb{N} \cdot j < k \leqslant i \rightarrow \ell_k \notin T_{FL})$

Given a set of fluents $\mathcal{F}$ over 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 ϕ and ψ are formulas, then so are $\neg\phi, \phi \vee \psi, \mathrm{X}\phi, \phi \mathrm{U} \phi$.

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

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

$$
\begin{aligned}
\pi, i \vDash \neg\phi \quad &\text{iff.} \quad \neg(\pi, i \vDash \phi) \\
\pi, i \vDash \phi \vee \psi \quad &\text{iff.} \quad (\pi, i \vDash \phi) \vee (\pi, i \vDash \psi) \\
\pi, i \vDash \mathrm{X}\phi \quad &\text{iff.} \quad \pi, i+1 \vDash \phi \\
\pi, i \vDash \phi \mathrm{U} \psi \quad &\text{iff.} \quad \exists j \geqslant i \cdot \pi, j \vDash \psi \wedge \forall i \leqslant k < j \cdot \pi, k \vDash \phi
\end{aligned}
$$

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

We say ϕ holds in π, denoted $\pi \vDash \phi$, if $\pi, 0 \vDash \phi$. A formula ϕ holds in an LTS E (denoted $E \vDash \phi$) 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 architecture. 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-behaveness, 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

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.

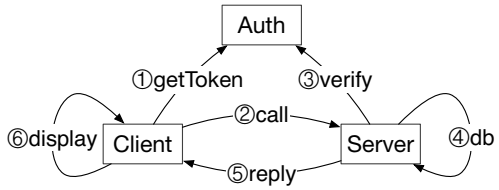
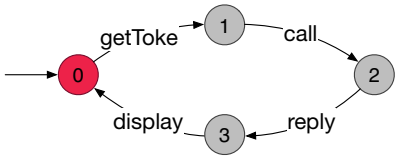
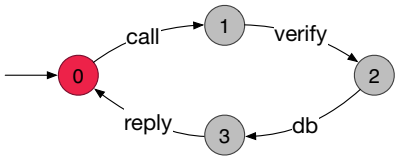


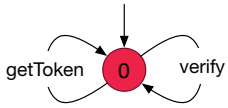
Fig. 1. Our running example



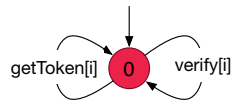
(a) Client LTS Cli_1



(b) Server LTS Ser_1



(c) Auth LTS $Auth_1$



(d) The i-th version $Auth_i$

Fig. 2. Components of our example

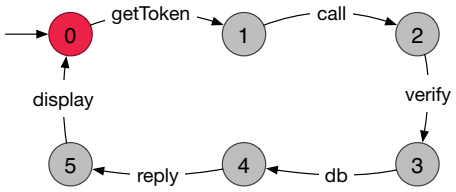


Fig. 3. $Cli_1 \| Ser_1 \| Auth_1$

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 form 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

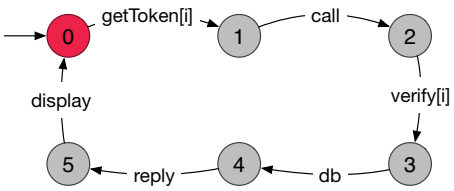


Fig. 4. The environment in DSU $E_1 = Cli \| Ser$

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 $Auth_1$ then the server must use $verify[1]$. The actual environment is shown in Figure 4.

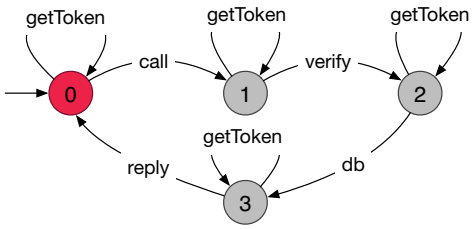
Assume we replace $Auth_1$ with $Auth_2$ after the client uses $getToken[1]$ but before the server uses the $verify[1]$. When the server needs $verify[1]$ but the $Auth_2$ has no such action, then deadlock happens. For simplicity, we assume that neither the environment nor the requirement changes. As 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 $Auth_1$ and $Auth_2$ 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.



$E = Ser \quad Auth$

Fig. 5. The Environment LTS $_1 \quad _1\| \quad _1$

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 $\Box(\phi \rightarrow \Diamond\psi)$, where neither ϕ nor ψ 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:

$$\{\ \text{DISPLAY\_AFTER\_CALL} = \Box(\dot{call} \rightarrow \Diamond\dot{display})\ \}$$

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

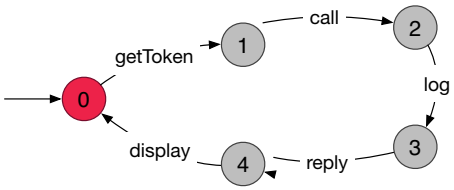


Fig. 6. Client LTS Cli_2

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 $S \vDash R$ to denote $S \vDash \bigwedge_{\phi \in R} \phi$ for short. In other words, when R occurs on the right side of $\vDash$, R stands for $\bigwedge_{\phi \in R} \phi$.

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

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

Definition 7. (DSU scheme) A DSU Scheme is a set $DS = \{(s,t) | s \in S_{P_1} \wedge t \in S_{P_2}\}$. $(s,t) \in DS$ 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 $D = (S_D, A_D, \Delta_D, s_D)$ is the LTS generated by applying the scheme into P_1 and P_2, where $S_D = S_{P_1} \cup S_{P_2}$, $A_D = \{dsu\} \cup A_{P_1} \cup A_{P_2}$, $\Delta_D = \Delta_{P_1} \cup \Delta_{P_2} \cup \{(s, dsu, t) | (s,t) \in DS\}$ and $s_D = s_{P_1}$.

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 FLTL to specify the period after a DSU as follows:

$$\text{AFTER\_DSU} = \langle \{dsu\}, \varnothing, false \rangle$$

Definition 9. (Correct DSU LTS, Correct DSU scheme) A DSU LTS D is correct iff. $D \| E_2$ is free of deadlocks and $D \| E_2 \vDash \Box(\text{AFTER\_DSU} \rightarrow R_2)$. 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, so we can make the parallel composition of E_2 and D and can interpret the traces in E_1.

The parallel composition of DSU LTS and E_2 is in Figure 8. The thicker purple edges indicate a trace which causes deadlock: $getToken, dus, log \nvDash$ 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.

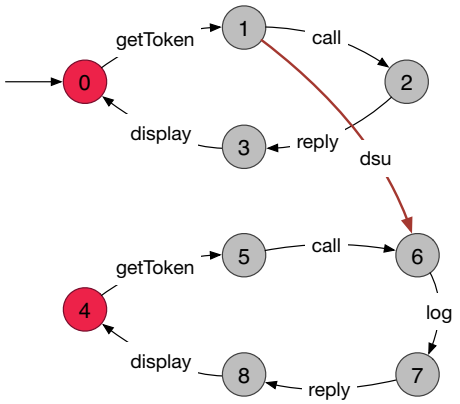


Fig. 7. A wrong DSU LTS in 1st scenario

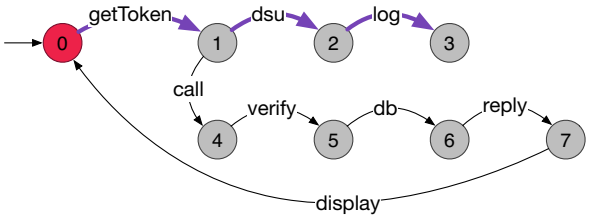


Fig. 8. $D \| E_2$ in 1st scenario

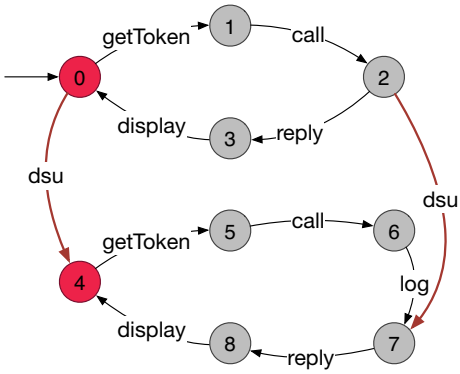


Fig. 9. A correct DSU LTS in 1st scenario

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.

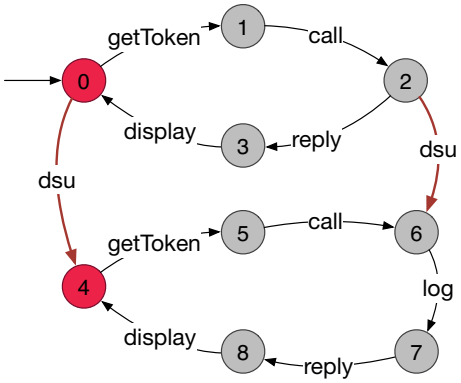


Fig. 10. A correct DSU LTS in 2nd scenario

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 $D\|E_2$, which has exact one i such that ℓ_i is dsu.

Intuitively, before the "*dsu*", the trace is executed as it is on the $P_1\|E_1$, and on the $P_2\|E_2$ after the "*dsu*". In other words, the i-length prefix of π is a prefix of a valid trace of $P_1\|E_1$ and $\ell_{i+1}, \ldots$ is a postfix of a valid trace of $P_2\|E_2$. A DSU trace π is correct iff. $\pi \vDash \Box(\text{AFTER\_DSU} \rightarrow R_2)$.

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:

$$\text{LOG\_BEFORE\_DISPLAY} = \Box(\neg(\dot{display} \wedge \neg\text{LOGGED}))$$

where,

$$\text{LOGGED} = \langle\{log\}, \{call\}, false\rangle$$

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.

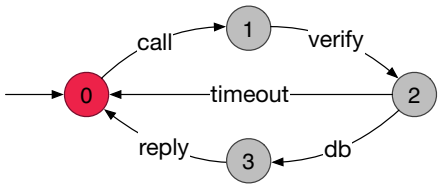


Fig. 11. Server LTS Ser_2

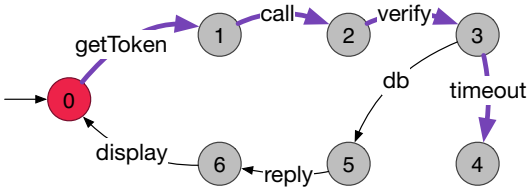


Fig. 12. Deadlock occurs in $Cli_1\|Ser_2\|Auth_1$

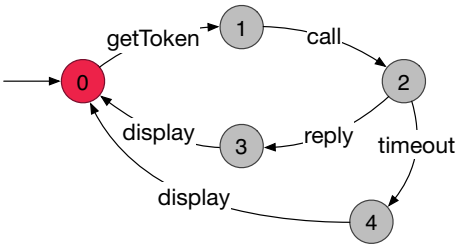


Fig. 13. New program LTS $P_2 = Cli_2$

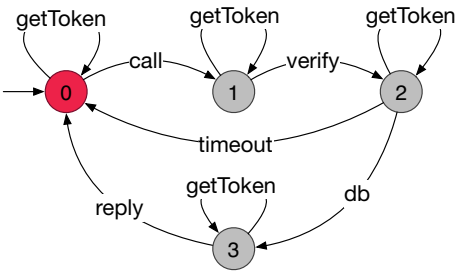


Fig. 14. Environment LTS $E_2 = Ser_2\|Auth_1$

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 $E_2 = Ser_2\|Auth_1$. The requirements are $R_1 = R_2 = \{\text{DISPLAY\_AFTER\_CALL}\}$. 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

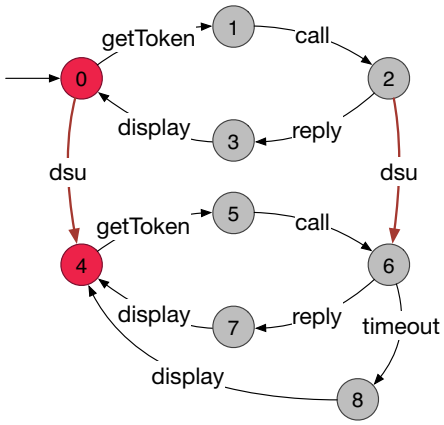


Fig. 15. A correct DSU LTS in 3rd scenario

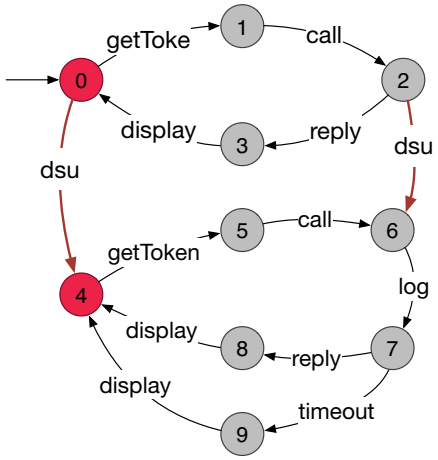


Fig. 16. A correct DSU LTS in 4th scenario

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 $E_1 = E_2 = Cli_1 \| Ser_1$ in Figure 4, $P_i = Auth_i$ in Figure 2(d) and $R_1 = R_2 = \{$Display_AFTER_CALL$\}$. The only possible DSU scheme of the Auth is in Figure 17 by our definition of DSU scheme. Figure 18 gives the LTS $D \| E_2$. $dsu, getToken[2], \ldots$ is a correct DSU trace but $getToken[1], dsu, \ldots$ 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.

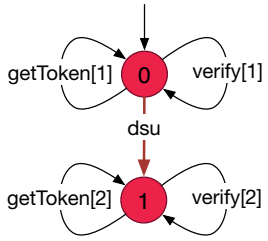


Fig. 17. DSU LTS D of $Auth_1$ and $Auth_2$

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 $verify[1]$. We can see from Figure 4 that the current state of the environment must be one of state 0/3/4/5 after one $verify[1]$ and before one $getToken[1]$. 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 dsu and it can observe other actions $Act \backslash \{dsu\}$. 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 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 *dsu* action and then

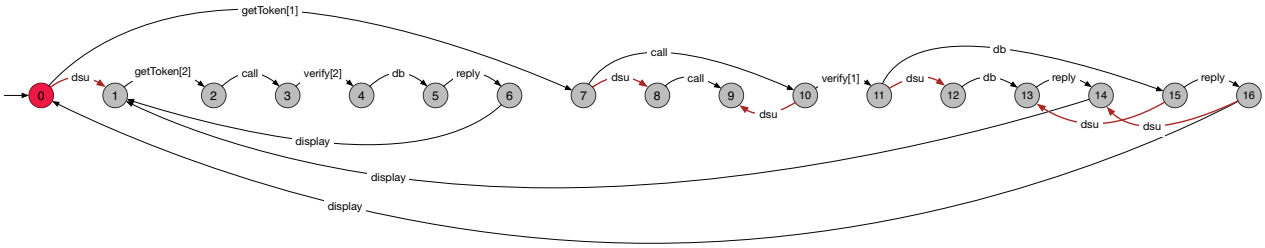


Fig. 18. $D \| E_2$ of Auth DSU

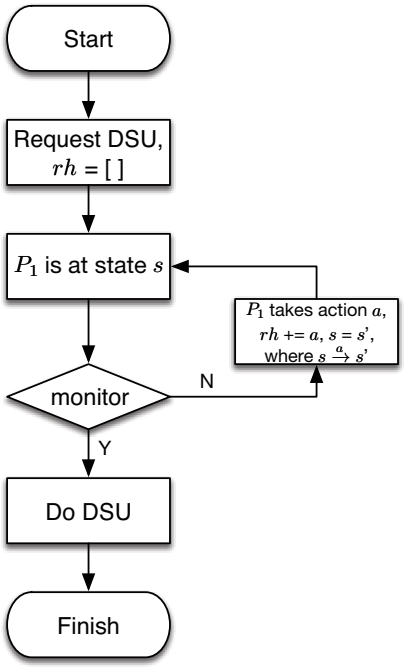


Fig. 19. Monitor-based DSU algorithm

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 π 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 π in the ideal monitor $Ctrl$, that $\pi[P_1]$ ends with rh implies that execution of π 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.

Algorithm 1 Monitor Generation Algorithm

Input:
The environment E_2
The DSU LTS D
The old system P_1
Output:
Monitor $DFA_M = (Q_M, \Sigma_M, \delta_M, s_M, F_M)$

$Q_M = \Sigma_M = \delta_M = s_M = F_M = \{\varnothing\}$
Sythesize the controller $Ctrl$.
if there is no $Ctrl$ **then**
  return $(Q_M, \Sigma_M, \delta_M, s_M, F_M)$
end if
$S^* = \{s | \exists t \in S_{Ctrl} \cdot (s \xrightarrow{dsu} t)\}$
$s_M = S_{Ctrl}$
$Q_M.add(s_M)$
$\Sigma_M = A_{P_1}$
$queue =$ new queue
$queue.enqueue(s_M)$
while queue is not empty **do**
  $S = queue.dequeue()$
  for each $act \in A_{S_1}$ **do**
    $S' = \{t | \exists s \in S \cdot (s \xrightarrow{act} t)\}$
    $S' = S' \cup \{t | \exists s \in S' \exists a \in A_{E_2} \setminus A_D \cdot (s \xrightarrow{a} t)\}$
    if $S' \notin Q_M$ **then**
      $queue.enqueue(S')$
      $Q_M.add(S')$
      if $S' \subseteq S^*$ **then**
        $F_M.add(S')$
      end if
    end if
    $\delta_M.add((S, act, S'))$
  end for
end while
return $(Q_M, \Sigma_M, \delta_M, s_M, F_M)$

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

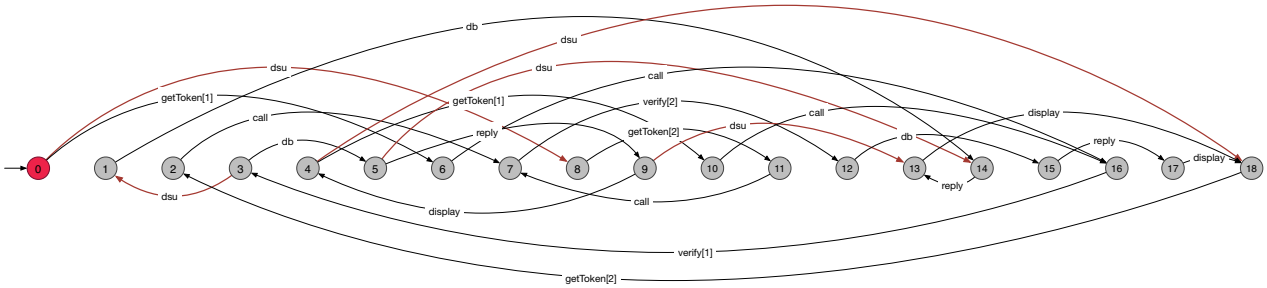


Fig. 20. The controller LTS $Ctrl$

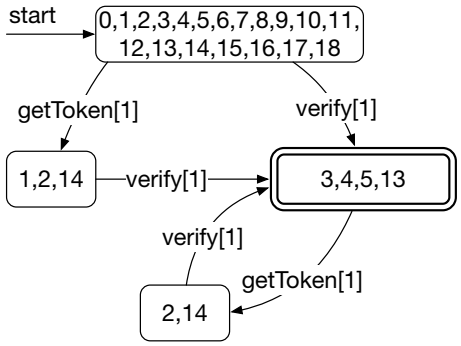


Fig. 21. The monitor DFA_M

$Ctrl$ 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 π in $Ctrl$, that $\pi[P_1]$ ends with rh implies that execution of π 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-

ciechowski *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 is 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. 61472177, 91318301, 61321491) of China.

References

[1] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol, "Practical dynamic software updating for c," in *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '06, pp. 72–83.

[2] S. Subramanian, M. Hicks, and K. S. McKinley, "Dynamic software updates: A vm-centric approach," *SIGPLAN Not.*, vol. 44, no. 6, pp. 1–12, Jun. 2009.

[3] 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.

[4] C. M. Hayden, K. Saur, E. K. Smith, M. Hicks, and J. S. Foster, "Kitsune: Efficient, general-purpose dynamic software updating for c," *ACM Trans. Program. Lang. Syst.*, vol. 36, no. 4, pp. 13:1–13:38, Oct. 2014.

[5] E. K. Smith, M. Hicks, and J. S. Foster, "Towards standardized benchmarks for dynamic software updating systems," in *Proceedings of the 4th International Workshop on Hot Topics in Software Upgrades*, ser. HotSWUp '12, pp. 11–15.

[6] T. Gu, C. Cao, C. Xu, X. Ma, L. Zhang, and J. Lü, "Low-disruptive dynamic updating of java applications," *Information and Software Technology*, vol. 56, no. 9, pp. 1086 – 1098, 2014, special Sections from "Asia-Pacific Software Engineering Conference (APSEC), 2012" and " Software Product Line conference (SPLC), 2012".

[7] X. Ma, L. Baresi, C. Ghezzi, V. Panzica La Manna, and J. Lu, "Version-consistent dynamic reconfiguration of component-based distributed systems," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pp. 245–255.

[8] J. Zhang and B. H. Cheng, "Model-based development of dynamically adaptive software," in *Proceedings of the 28th International Conference on Software Engineering*, pp. 371–380.

[9] K. N. Biyani and S. S. Kulkarni, "Assurance of dynamic adaptation in distributed systems," *Journal of Parallel and Distributed Computing*, vol. 68, no. 8, pp. 1097 – 1112, 2008.

[10] Y. Zhao, Z. Li, H. Shen, and D. Ma, "Development of global specification for dynamically adaptive software," *Computing*, vol. 95, no. 9, pp. 785–816, 2013.

[11] A. Pnueli and R. Rosner, "On the synthesis of a reactive module," in *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '89, pp. 179–190.

[12] R. M. Keller, "Formal verification of parallel programs," *Communications of the ACM*, vol. 19, no. 7, pp. 371–384, 1976.

[13] D. Giannakopoulou and J. Magee, "Fluent model checking for event-based systems," in *ACM SIGSOFT Software Engineering Notes*, vol. 28, no. 5, pp. 257–266.

[14] J. Magee and J. Kramer, *Concurrency: State Models And Java Programs*. John Wiley & Sons, 2006.

[15] N. D'Ippolito, "Synthesis of event-based controllers for software engineering," Ph.D. dissertation, Imperial College London, 2013.

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

[17] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," in *Proceedings of the 1999 International Conference on Software Engineering, 1999.*, pp. 411–420.

[18] N. D'Ippolito, D. Fischbein, M. Chechik, and S. Uchitel, "Mtsa: The modal transition system analyser," in *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pp. 475–476.

[19] S. Malek, G. Edwards, Y. Brun, H. Tajalli, J. Garcia, I. Krka, N. Medvidovic, M. Mikic-Rakic, and G. S. Sukhatme, "An architecture-driven software mobility framework," *Journal of Systems and Software*, vol. 83, no. 6, pp. 972 – 989, 2010, software Architecture and Mobility.

[20] J. Kramer and J. Magee, "Self-managed systems: an architectural challenge," in *Future of Software Engineering, 2007. FOSE '07*, pp. 259–268.

[21] D. Le Métayer, "Software architecture styles as graph grammars," *SIGSOFT Softw. Eng. Notes*, vol. 21, no. 6, pp. 15–23, Oct. 1996.

[22] P. Oreizy, N. Medvidovic, and R. N. Taylor, "Architecture-based runtime software evolution," in *Proceedings of the 20th International Conference on Software Engineering*, ser. ICSE '98, pp. 177–186.

[23] R. Allen, R. Douence, and D. Garlan, "Specifying and analyzing dynamic software architectures," in *Fundamental Approaches to Software Engineering*, ser. Lecture Notes in Computer Science, E. Astesiano, Ed., vol. 1382, pp. 21–37.

[24] J. S. Bradbury, J. R. Cordy, J. Dingel, and M. Wermelinger, "A survey of self-management in dynamic software architecture specifications," in *Proceedings of the 1st ACM SIGSOFT Workshop on Self-managed Systems*, ser. WOSS '04, pp. 28–33.

[25] C. M. Hayden, S. Magill, M. Hicks, N. Foster, and J. S. Foster, "Specifying and verifying the correctness of dynamic software updates," in *Proceedings of the 4th International Conference on Verified Software: Theories, Tools, Experiments*, ser. VSTTE'12, pp. 278–293.

[26] P. Wojciechowski and O. Rütti, "On correctness of dynamic protocol update," in *Formal Methods for Open Object-Based Distributed Systems*, ser. Lecture Notes in Computer Science, M. Steffen and G. Zavattaro, Eds., vol. 3535, pp. 275–289.