An Executable Model and Testing for Web Software based on Live Sequence Charts

Liping Li[1*]
[1]Computer and Information Engineering Institute, Shanghai Polytechnic University, Shanghai, 201209, China
*E-mail: liliping@sspu.edu.cn

Honghao Gao[2]
[2]Computing Center, Shanghai University, 200444, Shanghai, China
E-mail: gaohonghao@shu.edu.cn

Tang Shan[3]
[3]Computer and Information Engineering Institute, Shanghai Polytechnic University, Shanghai, 201209, China
E-mail: tangshan@sspu.edu.cn

***Abstract*—Static modeling is often difficult to understand when meet with complicated, large-scale Web software which has many unique characteristics. Aim at this problem, this paper proposes a method to create an executable model for Web software based on Live Sequence Charts (LSCs). The executable model can simulate the running of the system, which helps to find the inconsistency of the model in early development stage. Then the LSCs model is transformed to a symbolic automaton. Testing scenarios can be generated by traversing the automaton by depth-first search (DFS). Results showed test cases generated by this executable model are more effective than general model. We hope this method can do some help to the modeling and testing of the Web application.**

Keywords—Executable model; Web software; test scenario; Live Sequence Chart; symbolic automaton.

I. INTRODUCTION

Software testing is one of the most important means to guarantee software quality and improve software reliability. With the development of big data, cloud computing, Web software is becoming more and more complex. According to the Symantec Report, more than 60% of the software security vulnerabilities are about Web application, and these vulnerabilities may lead to various attacks and serious results [1]. How to ensure the quality and the security of Web applications have got much attention. But the characteristics of interaction, dynamic, heterogeneous, diversity and short development cycle etc. make the modeling and testing of Web software more complicated than before.

Static modeling is often difficult to understand for large-scaled Web software. Nowadays, the premise of most model-based testing is to suppose the test model is correct. If the test model itself is wrong, the test case generated by the model is also wrong. How to confirm the correctness of the model and how to verify them are still the hot topic in discussion.

Scenario-based specification language, such as Message Sequence Charts (MSCs), UML sequence diagram represent how system components, the environment, and users interact in order to provide system level functionality [2]. Live Sequence Chart (LSC) is a visual scenario-based modeling and specification language. It is an extension of MSC and UML 2 sequence diagram, but LSC are more expressive and semantically rich than them [3].

At present, there have been many research achievements on scenario-based modeling and testing, but most of the model cannot simulate the system's running dynamically.

Hussein et al. [4] proposed a method to describe the scenario by use case, they used classes to present the structure of the system, and state machines to model the behavior of the system. Massacci and Naliuka [5] use UML sequence diagrams and linear temporal logic to modeling behavioral of the system. But the semantics of UML sequence diagrams is not suited to model constraints like obligation, and prohibition. LSCs are well defined and have a strict formal semantics, and can be used at various stages of the software development and verification process [6]. Sven Patzina1 et al. [7] illustrated Live Sequence Charts are adequate for the specification of behavioral signatures. Paper [8] introduced Live Sequence Charts are a wonderful scenario-based language and introduced their tool Play-Engine. A lot of other work [9-13] presented Live Sequence Charts are well defined and possess a strict formal semantics. But, most work is investigating LSC model-checking problem. To the best of our knowledge, there is little paper focus on the modeling and testing of Web Applications by LSCs.

This paper proposed a new method to testing Web application. We first present how to construct an executable model for the Web application using use cases and Live Sequence Charts (LSCs). Then transform these LSCs model to symbolic automata. Finally generate test scenarios for Web application by traversing the automata. For the executable model, we first adopt hierarchical profile use cases to describe the high-level function requirement of Web software. Then use LSCs to model the detail behavioral requirements. Because LSCs can simulate the execution of the system dynamically, we can check how a running LSC affects the system behaviors in response to a set of external events.

The remainder of this paper is organized as follows: Section 2 describes the proposed approach to constructing the executable model for Web software and shows an example. Section 3 transforms LSCs to symbolic automaton and generates test scenarios. Section 4 discusses some related

978-1-5090-0806-3/16/$31.00 copyright 2016 IEEE
ICIS 2016, June 26-29, 2016, Okayama, Japan

work. Section 5 concludes the paper and highlights our future work.

II. The Construction of Executable Model

In this chapter, we show how to create the executable model for Web Application through an example. First, hierarchical use cases are applied to describe the high-level function requirements, and then LSCs is used to model the detail behavioral of the Web system.

A. The Hierarchical Use Case Model

A use case is an informal description of a collection of possible scenarios involving the system under test and its external actors. However, since use cases are high-level and informal by nature, they cannot be served as the basis for formal testing and verification. To support a more complete and rigorous development cycle, use cases must be translated into fully detailed requirements written in some formal language [4].

We use model driven approach to create the executable model for Web Application. In UML, use case diagram is used to specify system's high-level behavioral requirements. A use case diagram depicts actors, use cases, and relationships between them. Actors are external entities that interact with the system and use cases depict the function of the system. Actually, each use case is a sequence of actions that the system offers to its actors. LSCs are well defined and possess a strict formal semantics [8]. In this paper, we first apply hierarchical profile use cases to describe the high-level function requirement of Web application. Live Sequence Charts (LSCs) are used to model the detail behavioral specification of each use case.

The approach of using hierarchical profile use cases to describe the high-level function requirement is shown as below.

- From the users' perspective, Web application is divided into several main use cases according to actors; each of them completes a whole function of the actor.
- Divide the big use cases into a set of sub-use cases according to the function. The primary relationships between use cases are *include, extend, and generalization*. We extend the relationship between use cases with stereotype <<navigate>>, which means Web pages navigation.
- Repeat step 2 until the behavioral requirement of each use case in lower layer can be described easily by LSCs.

A Web Flight Reservation System (WFRS) is used as an example throughout the paper. The WFRS has three actors: customer, agency and administrator. Through this system, customers can search the flight information, reserve flight tickets, view his/her orders and return tickets etc. And agency also can search flight information, book tickets, and view customer's orders, modify, return or cancel orders. The administrator can add/delete/modify the information of customers and flights.

The top use-case diagram is shown as Figure 1. The function of customer, shown as figure 2, is divided into four use cases: Search Flight, Reserve Flight, View Orders and Return Tickets. The relationship between uses cases with stereotype <<navigate>> means the navigation of use cases. For the space limited, we omit other actors' use case diagram here.

The adoption of hierarchical profile use case diagrams to describe a complex Web application has the concise, comprehensive and integrated features.

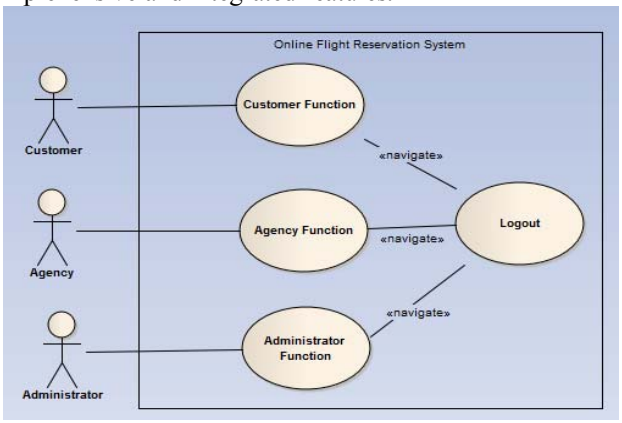


Fig 1. Top use case diagram

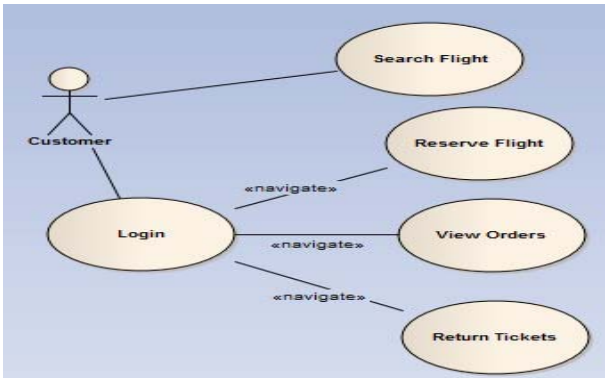


Fig 2. Use case diagram for customer

B. The Executable Model by LSCs

Live Sequence Chart (LSC) can serve to specify the behavior of either sequential or parallel systems, based on either centralized or distributed architectures, and they can be used to describe the interaction between processes, tasks, functions and objects [7]. LSC has universal chart and existential chart which can be used to describe mandatory behaviors or possible behaviors. A universal chart typically contains a pre-chart (denoted by a top dashed hexagon) and a main chart (denoted by a solid rectangle); pre-chart represents a precondition that has to be fulfilled before the main chart, if the pre-chart is satisfied, then the system is forced to satisfy the defined scenario in the main chart right below the pre-chart. Existential charts are more like MSCs and UML sequence diagrams. They are basic charts depicted in a dotted frame which are used in LSCs to specify sample interactions between the system and its environment and must be satisfied by at least one system run. An existential chart is usually used to specify a testing scenario that can be satisfied by at least one possible

system run. In this paper, we mainly focus on the universal chart.

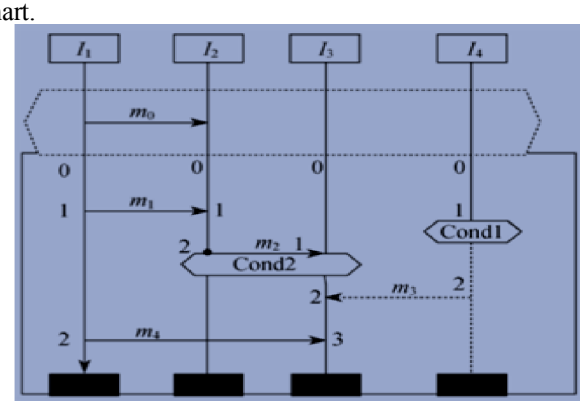


Fig 3. A Universal LSC

Objects / instance variables in LSCs are represented by a rectangular instance head and a vertical lifeline originating from each instance head. The lifeline represents the time dimension in the LSC with time progressing in the downward direction. Each lifeline of the object is marked with the number of points, which indicates that the event occurs or conditions, referred to as a *location*. According to the *location*, the sequence of events can be distinguished. As shown in figure 3, referred from paper [11], there are four instance variables/objects: I_1, I_2, I_3, I_4. Object I_1 has three location points $< I_1, 0>$, $< I_1, 1>$, $< I_1, 2>$. Communication between objects occurs with messages with the arrows representing the direction of communication. Condition is represented by hexagon, there are two conditions: Cond1 and Cond2. This paper assumes all messages in LSCs are *synchronous* message (filled arrowhead) where both the sender and receiver have to be ready for the message to be observed. There are some other important characteristics of live sequence chart, such as cut and temperatures etc.

The notion of temperatures have values of *hot* and *cold*, *hot* means elements must happen and *cold* is may happen [8]. For example, Object I_1, being at location l_1, sends a message m1 to I_2, who receives the message at location l_2, if the message and the locations are all hot, meaning that I_1 must send the message, the message must arrive, and I_2 must receive it. If the message and the location l_1 are hot, but location l_2 is cold, that is mean I_1 must send the message and the message must arrive. However, I_2 may decide not to receive the message. There are eight different cases, each indicating a different combination of temperatures for l_1, m_1 and l_2.

The detail introduction of LSC can be seen in [8]. An automatic tool Play-Engine has been provided to simulate scenario based behavior through play-in and play-out approach for LSCs.

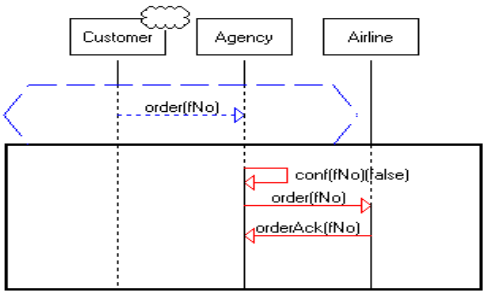
We use the universal LSCs to modeling the WFRS, shown as Figure 4. There are four objects in the example: Customer, Agency, Airline and Seller.

Customer and Seller are the external objects, denoted by waved clouds, which send the external/user inputs to the system. Objects *Agency* and *Airline* each has a state variable, *conf*, with the finite domain {false, true, abort} denoting whether the order has been initiated, confirmed or aborted respectively. There are two types of flight tickets, discount and non-discount. If the passenger reserved the non-discount one and at this point, the *Airline* has not confirmed the order, he/she can return the tickets; otherwise, he/she cannot return them. We introduce the parameter *fNo* to represent the ticket's type, *fNo*=1 denotes the non-discount tickets, 0 is the discount one.

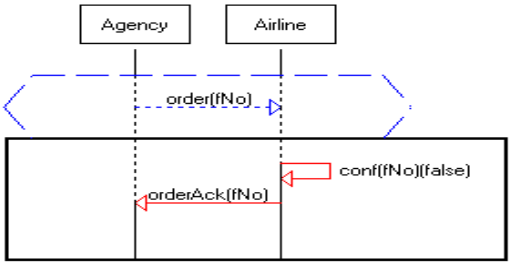
There are several scenarios in figure 4. Chart (a) (b) (c) describes the scenario-based behavior of a successful flight reserve. Chart (a) shows if a customer wants to reserve flight, he/she creates an order, then the order is initiated in *Agency* by setting *Agency.conf* false, sending a *Order* message to *Airline*, and waiting for the acknowledgment; chart (b) says that if *Airline* receives an order request, it will set *Airline.conf* false, and then reply to *Agency* an acknowledgment; chart (c) shows that *Agency* receives an order confirmation from *Airline*, and then sets its state variable *Agency.conf* true. Chart (d) (e) (f) represents the scenario of aborting the order. Chart (d) describes if the customer wants to abort the order, and at that point if the order has not been confirmed yet, an abort message will be forwarded to *Airline*, or otherwise, an appropriate message according to order status will be send to the *Customer*. The three stacked rectangles in the main chart is a select-case construct, where each hexagon contains a condition. Chart (e) says that if *Airline* receives an abort request, and at the same time, if *Airline.conf* is still false and the reserved tickets are non-discount, then change the value of *Airline.conf* to *abort* and reply with an abort confirmation message; otherwise, reply with an abort deny message; chart (f) expresses that if *Airline* receives an order confirmation from *Seller* directly, and at that point, if *Airline.conf* is still false, then sets it true, or otherwise, an appropriate message according to order status will be send to the *Seller*. Chart (g) shows that *Agency* receives an abort deny from *Airline* and chart (h) shows that *Agency* receives an abort confirmation from *Airline*, and then sets its state variable *Agency.conf* abort.

As we can see, the message and the locations of this example are all *hot*, denoted by the solid lines, meaning that *mandatory* behavior that things must move on. For example, in chart (a), in pre-chart, Customer want to order flight, he/she sends a message to *Agency*, in the main chart, the order is initiated by *Agency* setting *Agency.conf* false, that is mean the order is new, then the *Agency* sending the *Order* message to *Airline,* because the message and the location are all *hot, Agency* must send the message *order(fNo)* to *Airline*, and the message must arrive, and *Airline* must receive it.

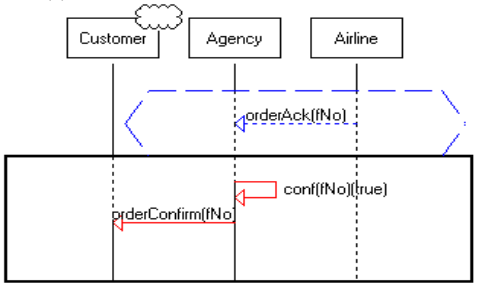
We can execute and monitor the LSCs in the Play-Engine. The Play-Engine is a tool with "play-in/play-out" approach developed by David Harel and Rami Marelly [8]. Play-out allows a convenient way to debug requirements at an early stage and to detect problems in the design. But Play-Engine is only a research-level tool, not a commercial product. So the maintain is not as good as commercial one.



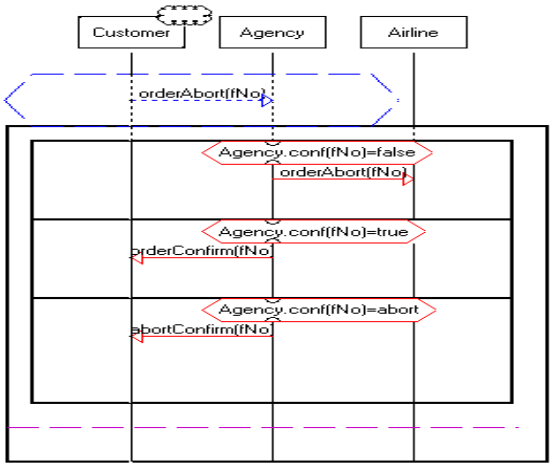
(a) Agency creates an order



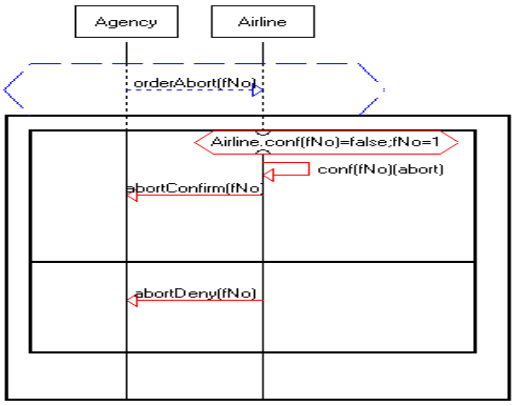
(b) Airline receives an order



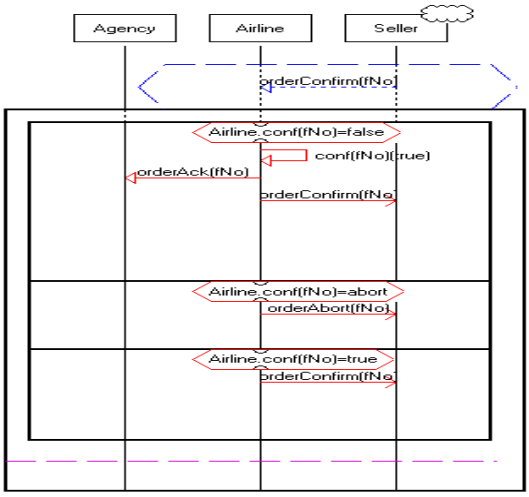
(c) Agency receives an order confirmation



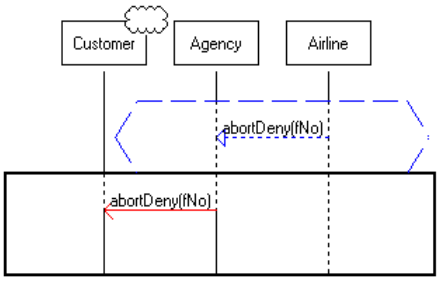
(d) Agency aborts an order



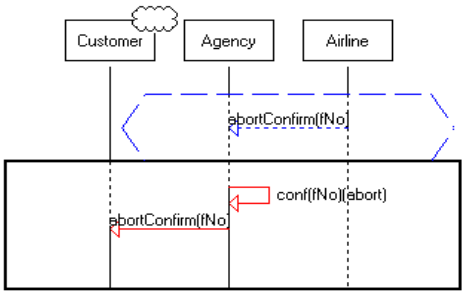
(e) Airline receives an order abort



(f) Airline confirms the order



(g) Agency receives an abort deny



(h) Agency receives an abort confirmation

Fig 4. An LSC example--Web Flight Reserve System

III. TRANSFORM LSCS TO SYMBOLIC AUTOMATON AND GENERATE TESTING SCENARIOS

A. Transform LSCs to Symbolic Automaton

Previous work shows that the LSC-to-temporal logic and LSC-to-automata translations can be automated and formalized for the LSC language [6]. In this paper, we convert the LSCs model to corresponding symbolic automata based on paper [10]. The symbolic automaton for LSCs is basically a Buchi automaton where the transitions are labeled by events, messages ([parameter variables]) or Boolean conditions.

The main difference between the LSC and the MSC language is that LSCs introduce modalities for whole charts, locations on instance lines, and LSC elements [10]. Referred to paper [10], we describe the abstract syntax and semantics of LSCs by a new formalization. We only concentrate on the universal chart and the main characters of LSCs, called it core LSC.

Definition 1 (Core LSC). A live sequence chart is a tuple $Ls = (O, L, E, M, \prec, C)$ where O is a set of instances variables, L is a set of lifelines, E is a set of events appearing in Ls and M is a set of messages, $\prec$ is the partial order on L, C is the runtime locations of the instances of Ls.

Definition 2. Let Ls be a LSC, L is a set of lifelines, a lifeline $l \in L$ is a sequence of events $l = (e_1, e_2, ..., e_j, ..., e_n)$, let $E(Ls)$ be the set of events appearing in Ls. The chart Ls induces a partial order relation $\prec$ on $E(Ls)$ as follows:

1) $\forall (e_1, e_2, ..., e_j, ..., e_n) \in E(Ls)$ and $1 \le j < n$, it holds that $e_j \prec e_{j+1}$; and

2) $\forall m \in M$, if (m, s) and (m, r) $\in E(Ls)$, then (m, s) $\prec$ (m, r). Where (m, s) denotes the event of sending m, and (m, r) denotes the event of receiving m.

Definition 3 (Cut). Let Ls be a LSC and O be the instance variables of Ls. $\forall i \in O$, DOM (Ls, i) = $\{l_0, l_1, ..., l_{max(i)}\}$ is the point set of position for instance variable i from the first point l_0 to the last point of $l_{max(i)}$. DOM (Ls) = $\{(i, l) \mid i \in O \wedge l \in$DOM ($Ls$, i)$\}$ is a dual set of all instances of Ls and its mapping location point.

Actually, the running of LSCs is a Cut sequence. Each Cut in sequence is the successor of the Cut before. Cut represents a mapping set of the current location points of all instance variables in a LSC, that is, a set of mapping of each instance to a location point at which Ls is running. By Cut, we can acquire the progress along an instance line. So we can use Cut to represent as a state, then a cut sequence can be considered as a series of state transition.

Definition 4 (Symbolic Automaton of an LSC). The Symbolic Automaton of a LSC Ls, denoted by SA, is a tuple $SA = (S, s_0, E, \Delta, F)$ with S = Cut (Ls), s_0 is the initial state, E is a set of all system events including external events, internal messages among system objects, conditions or hidden events defined in LSCs, $\Delta \subseteq S \times (\varepsilon \cup E) \times S$ is the set of allowed transitions, $F \subseteq S$ is final states set.

Given a current state $s \in S$ and a system event $e \in E$, the transition $\Delta(s, e)$ returns an updated state after processing one or more of actions.

According to definitions above, we can transform the LSCs model of figure 4 to a symbolic automaton, shown as figure 5. In figure 5, the initial state is s0 and the final states are s6, s9, s13 and s16, which represented by double circle. The formal symbolic automaton for Web Flight Reserve System is shown as below:

- $SA = (S, s_0, E, \Delta, F)$;
- S = {s0, s1, …, s16}, s0 =(Customer, l_0) is the initial state, F={s6, s9, s13, s16};
- E={order(fNo), Conf(fNo)[false], orderAck(fNo), orderConfirm(fNo), orderAbort(fNo), Agency.Conf(fNo)[true], Agency.Conf(fNo)[false], Agency.Conf(fNo)[abort], orderAbort(fNo), abortDeny(fNo), abortDeny(fNo), Conf(fNo)[abort], abortConfirm(fNo)};
- Δ: Δ(s, order(fNo))=s1,
  Δ(s1, Conf(fNo)[false])=s2,
  Δ(s0, orderAbort(fNo))=s7, …,
  Δ(s15, orderConfirm(fNo))=s16.

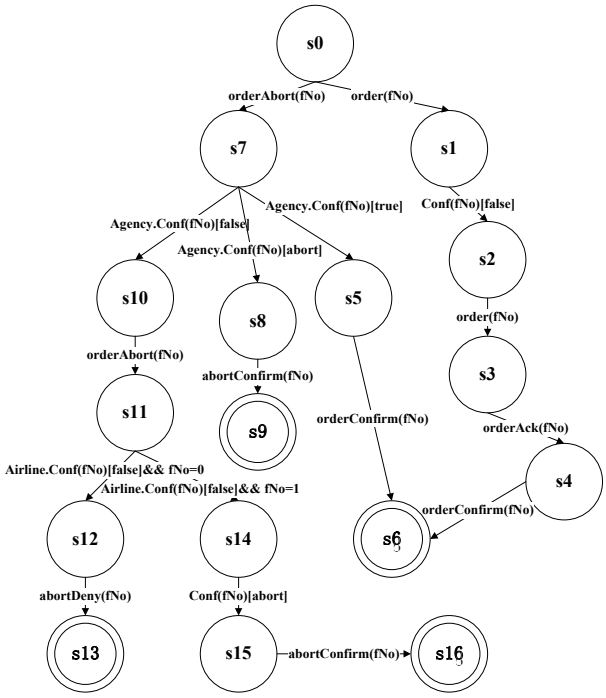


Fig 5. The converted symbolic automaton

B. Generate Testing Scenarios

We traversing the symbolic automaton use **depth-first search (DFS)** method. Start from the initial state s0, and explores as far as possible along each branch before reach one of the final state, collect the sequence of transitions/states from the initial state to the final state, this transitions/states sequence is one test scenario for the Web system. Repeat the steps above until all the states are traversing.

For the Web Flight Reserve System, we can obtain five test scenarios shown as below:

Ts1: s0, s1, s2, s3, s4, s6

Ts2: s0, s7, s5, s6

Ts3: s0, s7, s8, s9

Ts4: s0, s7, s10, s11, s14, s15, s16

Ts5: s0, s7, s10, s11, s12, s13

Among the five test scenarios, Ts1 is the scenario for a successful flight reserve; Ts2 is a scenario for the customer wants to abort the ticket, but the ticket is confirmed, so he/she cannot return the tickets. Ts3 is the scenario that the customer wants to abort the subscribing and the ticket is not confirmed, so the Airline cancel the order; Ts4 is the scenario that the subscribing is success, but the customer want to return and this is a non-discount ticket, so the Airline return the ticket. Ts5 is the scenario that the book is success and the customer want to return the ticket, but this is a discount ticket, so the customer cannot return the ticket. We can get test cases by substituting specific data for symbols in test scenarios.

IV. RELATED WORK

Recent years, there are a lot of papers [6-13] investigated live sequence chart, most of them are about the model-checking problem by transforming LSCs to an automaton. Paper [6] focused on the problem of formally verifying systems against LSC specifications. Paper [7] illustrated live sequence chart is adequate for the specification of behavioral signatures of the embedded systems. Paper [8] introduced a lot of information about LSCs and their tool Play-Engine. Our work use Play-Engine to model the Web application. Paper [9] first introduced model-checking LSCs against system models. Paper [10] gave a well-formedness of LSCs in terms of concrete syntax and the semantics-giving automata. Paper [11] used LSCs to describe scenario based requirement. They presented three criteria for generating and extracting the safety, reachability and liveness properties from LSC. Paper [12] proposed an algorithm that transforms LSCs to automata, which enables the verification of communication protocol implementations. Paper [13] proposed a method to verify systems against LSC specifications by transforming the LSC to a positive automaton. But as we know, there is little paper concentrate on the modeling and testing Web application by LSCs.

V. CONCLUSIONS AND FUTURE WORK

In view of the unique characteristics of Web software, this paper proposes a method to create an executable model for Web software using live sequence charts (LSCs). The executable model can simulate the running of the system which helps to find the inconsistency of the design model and requirement specification. In order to generate test scenarios from the model, we transform the LSCs to symbolic automaton. Traversing the symbolic automaton by depth-first search (DFS), we can obtain test scenarios for Web software. Results showed test cases generated from our executable model are more effective than those generated from general model. So, it can improve the quality of Web software and reduce the security vulnerabilities in some degree. Our future work is to apply this method on some large Web system and discuss how to control the state explosion problem for the symbolic automaton of the large system.

Acknowledgment

This paper is supported by National Natural Science Foundation of China (NFSC) under Grant No. 61502294, The Natural Science Foundation of Shanghai under Grant No. 15ZR1415200, The Key Disciplines of Computer Science and Technology of Shanghai Polytechnic University under Grant No.XXKZD1604, and Internet Technology of CERNTER under Grant No.NGII20150609.

References

[1] Symantec Internet security threat report trends for January-June 07, Volume XII[R]. Cupertino. Symantec Corporation.2007.

[2] G. Sibay, S. Uchitel, and V. A. Braberman. Existential Live Sequence Charts Revisited, ICSE'08, Leipzig, Germany. pp. 41-50, 2008.

[3] Werner Damm and David Harel, "LSCs: Breathing Life into Message Sequence Charts". Proc. 3rd IFIP Int. Conf. on Formal Methods for Open Object-based Distributed Systems, 1999, pp. 293–312.

[4] Hussein, M., Zulkernine, M.: UMLintr: A UML Profile for Specifying Intrusions. Ins: 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems, ECBS 2006, pp. 8–288. IEEE, Los Alamitos (2006)

[5] Massacci, F., Naliuka, K.: Towards Practical Security Monitors of UML Policies for Mobile Applications. In: Proc. of IEEE POLICY 2007, pp. 278 (2007)

[6] Rahul Kumar1 Eric G Mercer. Improving Translation of Live Sequence Charts to Temporal Logic. Electronic Notes in Theoretical Computer Science. 2006

[7] Sven Patzina, Lars Patzina, Andy Schu¨rr. Extending LSCs for Behavioral Signature Modeling. SEC 2011, IFIP AICT 354, pp. 293–304, 2011.

[8] David Harel and Rami Marelly, Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine. Springer-Verlag, 2003.

[9] Bunker, A., Gopalakrishnan, G., Slink, K.: Live sequence charts applied to hardware requirements specification and verification: A VCI bus interface model. Software Tools for Technology Transfer 7 (2004) 341–350.

[10] Bernd Westphal and Tobe Toben. The good, the bad and the ugly: Well-formedness of live sequence charts, FASE 2006, LNCS 3922, pp. 230–246, 2006.

[11] Dai Yu-ting, Miao Huai-kou, Mei Jia, and Hao Hong-hao. Property Extraction Based on LSC Model Checking, Journal of Shanghai University (Natural Science), Vol. 18, No2, April. 2012, pp. 156-162

[12] R. Kumar and E. Mercer: Improving live sequence chart to automata translation for verification. Electronic communications of the EASST, 2008.

[13] J. Klose, T. Toben, B. Westphal, and H. Wittke: Check It Out: On the Efficient Formal Verification of Live Sequence Charts. 18th International Conference on Computer Aided Verification (CAV), pp. 219–233, 2006.