Int J Software Informatics, Volume 6, Issue 2 (2012), pp. 271–306
International Journal of Software and Informatics, ISSN 1673-7288
©2012 by ISCAS. All rights reserved.
E-mail: ijsi@iscas.ac.cn
http://www.ijsi.org
Tel: +86-10-62661040

Secure System Modeling: Integrating Security Attacks with Statecharts

Omar El Ariss[1] and Dianxiang Xu[2]

[1] (Department of Computer Science and Mathematical Sciences,
The Pennsylvania State University, Harrisburg, PA, USA)

[2] (National Center for the Protection of the Financial Infrastructure,
Dakota State University, Madison, SD, USA)

Abstract Software security is becoming an important concern as software applications are increasingly depending on the Internet, an untrustworthy computing environment. Vulnerabilities due to design errors, inconsistencies, incompleteness, and missing constraints in software design can be wrongly exploited by security attacks. Software functionality and security, however, are often handled separately in the development process. Software is designed with the mindset of its functionalities and cost, where the focus is mainly on the operational behavior. Security concerns, on the other hand, are often described in an imprecise way and open to subjective interpretations. This paper presents a threat driven approach that improves on the quality of software through the realization of a more secure model. The approach introduces systematic transformation rules and integration steps for integrating attack tree representations into statechart-based functional models. Through the focus on the behavior of an attack from the perspective of the system behavior, software engineers can clearly define and understand security concerns as software is designed. Security analysis and threat identification are then applied to the integrated model in order to identify and mitigate vulnerabilities at the design level.

Key words: software security; attack trees; threat modeling; system modeling; statecharts

El Ariss O, Xu DX. Secure system modeling: Integrating security attacks with statecharts. *Int J Software Informatics*, **Vol.6, No.2 (2012): 271–306. http://www.ijsi.org/1673-7288/6/i121.htm**

1 Introduction

Computer security is concerned with identifying and countermeasuring security threats at three different layers: application layer, networking layer, and operating system layer. Attacks that target the application layer cannot be mitigated through the use of secure networking communication or secure operating system services, such as firewall or encryption. This paper concentrates on software security, or security at the application layer.

Software security is concerned with the concept of understanding the vulnerabilities of the software system and preventing the system from being compromised. Software vulnerabilities account for 70% of the vulnerabilities found in business applicat-

Corresponding author: Omar El Ariss, Email: Oelariss@psu.edu
Received 2011-03-30; Revised 2011-06-30; Accepted 2011-11-20; Published online 2012-03-19.

Copyright by Institute of Software, Chinese Academy of Sciences. All rights reserved

ions[1]. Vulnerabilities or security holes in software are the result of either bugs in the implementation or flaws in the specification and design. Even when software is implemented correctly, it is still susceptible to security attacks when its design is erroneous. Current increase in software applicability and functionalities add more complexity to the tasks and requirements specification of software. As software becomes more complex, the chance of design errors and ambiguities increases and as a result system vulnerability to security attacks increases.

The consideration of security concerns at the design level suffers from the same problems and difficulties of software requirements that are non-functional in nature. These requirements are usually expressed and represented during the requirements phase in an ambiguous, imprecise description. During the design phase the focus is mainly on the precise description and modeling of functional requirements, while security requirements or concerns are neglected. As a result, software engineers design and build the system with a subjective, limited, and in most cases lacking a point of view on security. There is a need for these security concerns to be depicted in a form that allows the measurement and evaluation of security as the software is developed. Unless security concerns are addressed during the design phase, the software is inevitably vulnerable.

Threat modeling has been considered critical to the development of secure software[2−3]. As Howard and LeBlanc argued, "you cannot build a secure system until you understand your threats!"[4]. This modeling technique derives and ranks threats from Data Flow Diagrams (DFD) of a system, then resolves these threats through mitigation. It provides a structured approach to determine threats without only relying on the security architect's previous experience of security attacks but on the solid understanding of the system architecture. Threat modeling usually consists of four steps: modeling system functions, specifying security threats, ranking threats for risk analysis, and mitigating threats[5]. In this process, software is first described using DFD diagrams. Security threats are then represented as attack trees[6] that depict different potential attacks against a system. Threat mitigations or security requirements can then be derived from those attack trees.

The threat modeling approach advocates the incorporation of security consideration while designing the system[5]. It uses separate models to represent threats and system behavior. Attack trees are constructed by security architects. These models are intended to be simple in structure and easily understood. This allows experts from different discipline to communicate and understand security concerns through a model that is familiar to most of them. Yet, these models represent security attacks at a high level of abstraction. This makes attack trees insufficient for software engineers to clearly understand security attacks and their impact on the software they are designing. On the other hand, software engineers construct system models that depict software functionalities at a detailed level in order for them to clearly understand, carefully analyze, and validate the correctness of the design. As a consequence, the use of separate models, one for threat modeling and another for behavioral modeling, brings with it some drawbacks:

- An attack tree does not describe how the system behaves while the threat is occurring. Instead, it describes different ways an attacker can perform the threat by following a step by step procedure. Looking at an attack tree gives no clear

Copyright by Institute of Software, Chinese Academy of Sciences. All rights reserved

idea which system component/s were exploited, and how the behavior of the software affects or gets affected by these security threats.

- The system model emphasizes the functional behavior of the system. Thus, software engineers only understand the system from its intended behavior and do not clearly understand how a system might behave wrongly due to design flaws or lack of constraints.

- Semantic differences between the two models might result in different interpretations by software engineers. They might face difficulties in precisely defining threat attacks due to the dependency of the attacks on the environment and the attacker's capability.

The main hindrance in software security is that software developers lack the understanding of security concepts. Software is designed with the mindset of its functionalities and cost, where the focus is on the operational behavior, while security concerns are neglected or marginally considered. As a result, software engineers build the software while lacking the knowledge about security and its effect on the system. Therefore, a specification model that covers security threats is crucial in ensuring that security is not only considered at an early stage during the design phase but also throughout the software life cycle. This will allow both software engineers and security experts to base their understanding of the system and its security issues on one model rather than multiple models with different semantics.

This paper presents a threat driven approach that allows the modeling of security attacks together with the system's behavior. The proposed approach builds on our previous work in Ref. [7] and Ref. [8], where threats are modeled using statecharts and attack trees are integrated into system models, respectively. Each security attack is integrated into the system statechart resulting in an integrated model. The main purpose of this paper is to increase the security awareness of software engineers by modeling the dynamic behavior of security attacks and integrating it with the functional specifications. From this model, and through the focus on the behavior of an attack against the system behavior, it is possible to verify system functionalities in order to identify and mitigate vulnerabilities and remove ambiguities, errors, conflicts and inconsistencies in the system design. Any future modifications or additions to the system can be analyzed through this model to check whether they cause new vulnerabilities.

The contribution of this paper is a new modeling approach that integrates functionality and security concerns. The integrated model will not only depict the different ways an attack can be enacted but will also describe the attack through the behavior of the system. It facilitates the identification and mitigation of vulnerabilities. It is amenable to look at the statechart and clearly identify the components that represent threats from the ones that represent system functionalities.

The remainder of this paper is organized as follows. Section 2 discusses related work. Section 3 gives a brief background on attack trees and statecharts. Section 4 introduces the rules for transforming attack tree gates. Section 5 describes the integration process. The shopping cart case study is presented in Section 6 and we conclude in Section 7.

Copyright by Institute of Software, Chinese Academy of Sciences. All rights reserved

2 Related Work

Poor design is a major source of software security risk[9]. The process of identification, description, modeling and analysis of security threats is fundamental for access control systems[10], intrusion detection systems[11], and penetration testing[12]. Security concerns can either be represented graphically or through written documentation. Graphic representations of security threats and attacks range from the use of fault trees[13], attack trees[6], misuse cases[14−16], anti-goals[17] and Petri net[18,19] models. Written documentation of security threats can be represented through the use of vulnerability databases[20], evaluation criteria such as Ref. [21] or security patterns[22]. We focus in this paper on modeling the dynamic behavior of a security attack and the behavior of the system with its components as they undergo an attack. In comparison to our proposed approach, the aforementioned graphical representations either capture security concerns at a high level of abstraction that does not permit a deep understanding of threats and system vulnerabilities or at an abstraction, such as Petri nets, that is not intended for modeling the behavior of threats and their interaction with the functionalities of the system.

Security concerns can either be modeled to depict the attackers' perspective (what they are trying to achieve) or to describe generic vulnerabilities (applicable to different contexts and software applications) and their solutions. Although models that concentrate on the general description of vulnerabilities and how they can be mitigated are important for security analysis, these types of representations do not allow the discovery of unprecedented threats or attacks. Modeling techniques that concentrate on the intents of an attacker, such as threat modeling, provide the specialists with a deeper understanding of how an attack is made and therefore can help them detect new threats. Our work aims at improving the knowledge of security concerns as software is deisgned an implemented. Therefore we focus on security attacks that reflect the standpoint of an attacker.

There has been some work that introduced security into UML models using Aspect-Oriented Modeling (AOM). This work focused on security solutions. Ray et al. in Ref. [23] introduced an approach that weaves access control requirements (aspects), represented as patterns, into the UML model. While in Ref. [24], the authors presented an AOM approach for weaving security solutions into the software. They start by specifying the security requirements of the system and then model them as UML profiles. From these requirements, security solutions are derived and represented as UML profiles for aspect-oriented security modeling. Both of the above approaches do not represent the security concern graphically. On the other hand, Pavlich-Mariscal et al.[25] extend the work of Ref. [23] by including security diagrams to represent both access control policies and security features.

Jurjens in Ref. [26] introduced UMLsec, an extension to UML that incorporates security requirement analysis to the design of software. In UMLsec, statecharts are used to include security requirements as guards and assertions. Xu and Nygard[19] proposed a threat driven approach that models and verifies secure software applications. In their approach, system functionalities are specified from use cases while security threats are elicited from misuse cases. Both the functionalities and security concerns are represented as Petri nets. The deduced threat mitigations are then modeled as Petri net-based aspects. Kong and Xu[27] introduced a UML framework to

Copyright by Institute of Software, Chinese Academy of Sciences. All rights reserved

verify security threats. System behavior is depicted as UML statecharts while security threats are represented as sequence diagrams. Security threat verification is then conducted on the functionality of the system through the use of a graph transformation system. Compared to the above research work, our current approach is not only concerned with the introduction of security concepts for the validation and verification of functional specifications. Our proposed approach represents the dynamic behavior of security attacks as statechart notations and integrates the attacks into the functional behavior of the system.This integrated model allows a better understanding of security issues through a semantically well defined representation and through the analysis of both security and functional requirements.

Our previous work[28] presented a technique that integrated hazards, depicted using fault trees[29], into the system statechart. Although attack trees are simpler in form (due to the limited types of gates used), they are more complex to integrate. This is because an attack tree is not intended to describe the behavior of the system while the threat is occurring. Instead, the attack tree represents the point of view of an attacker by describing how an attacker can perform an attack following a step by step format. On the other hand, fault trees represent hazards which are the product of the behavior of the system; that means all the nodes are part of the functional behavior. As for attack trees, many of the nodes might stand for external behavior. Another reason is that attack trees are not semantically well defined[18,30].

Usually, software engineers mainly concentrate on the functionalities of the system overlooking features that contribute to the quality of the software they are designing. One of the important attributes of quality is software security. One of the benefits for modeling the dynamic behavior of an attack at a low level of abstraction is to improve and strengthen the software engineers' concept of security as the software is being designed. This emphasis on improving the software engineers' grasp of security concerns as the software is being designed is not stressed at all in the aforementioned research work.

3 Background

3.1 Attack trees and attack tree semantics

Attack trees[6] have been widely used to specify security threats. An attack tree is a hierarchical structure that depicts the different ways a security attack can occur against a system. The decision making process is composed of a logical gate and its input. The gate type reflects the kind of decision the attacker is making. Gate inputs reflect the option given or steps needed for an attacker to accomplish something. Attack trees make use of two types of gates to depict the attacker's decision: conjunctive (AND) and disjunctive (OR) gates. AND gates are used to represent the set of actions that should be done together in order to achieve the parent node. OR gates are used to represent the set of different options, where performing at least one of these options will achieve the parent node. Figure 1 shows an example of an attack tree, pointing the two different types of gates.

Copyright by Institute of Software, Chinese Academy of Sciences. All rights reserved

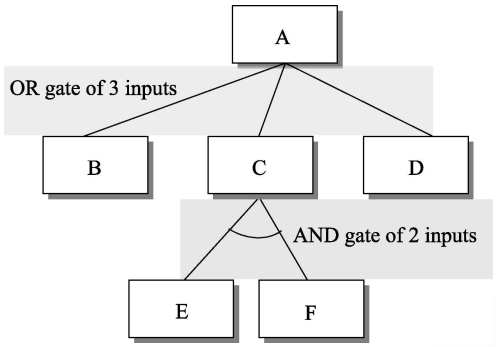

Figure 1. A general representation of an attack tree

An attack tree describes, from the perspective of an attacker, the possible ways an attack can be achieved successfully against a system. The single top node or the root node of an attack tree represents the attacker's ultimate goal or the targeted security attack. Child nodes represent sub-goals, i.e., possible ways to achieve the goal of their parent node. The root node is composed of one or more child conditions. These child conditions can either have a conjunctive relationship or a disjunctive relationship. Subsequently, child conditions can also be sub-goals that are composed of other child conditions that model the steps required for achieving each respective sub-goal. Child conditions that do not have further refinements are considered to be leaf nodes.

The semantics of an attack tree can be deduced from the semantics of the root node. While the semantics of the root node is defined from the semantics of its intermediate nodes, gates and edges. Similarly, an intermediate node is defined by the semantics of its leaves, edges, and gates in the sub-tree where the intermediate node is considered as a root. Therefore, the semantics of the tree structure can be defined from the semantics of its edges, gates and leaf nodes.

3.2 Statecharts

Statechart, introduced by David Harel[31], is a behavioral model that depicts the functional specifications of a system. Statecharts are extensions to state machines with the introduction of hierarchy and orthogonality. They are used to represent the behavior of complex systems, such as reactive ones, in a clear and understandable form without suffering from explosion in the number of states and edges. Statecharts were not intended to be a mere specification tool but a formalism or a language that can be compiled and executed[32,33]. Although statecharts are effective in representing the dynamic behavior of a system, they lack the capability of qualitative and quantitative analysis of the behavioral properties[34].

A statechart is composed of states, transitions and events. States represent the components and subcomponents of a system. Transitions between the states depict the system and subsystem interactions, while events and actions trigger these transitions. Orthogonality is supported through the use of parallel states separated by dashed lines (and-states). Communication between orthogonal parts is done through the use of actions and through broadcasting of events. Hierarchy is supported by allowing states to encompass other states (or-states). We have chosen statecharts

Copyright by Institute of Software, Chinese Academy of Sciences. All rights reserved

because they are a popular formalism for representing the functional behavior of the system and because they have descriptive capabilities to depict the behavior at a detailed level.

4 Gate Transformation

This section describes the transformation of attack tree representations, which are static high level abstractions, into statechart notations, which are dynamic low level behavior. The outcome will be a threat representation that is component based. In the following subsections, we will describe the statechart notations that are needed to successfully depict a security threat. We will use states, transitions and events to represent the dynamic behavior of an attacker. In this context, a state represents the accomplishment of a certain step out of a set of steps needed to successfully complete an attack. State transitions describe the progresses towards the achievement of the attacker's ultimate goal. Events trigger these transitions, thus depicting the interactivity of the model. Also, we use conditional entrances in order to model conditional behavior in statecharts. These are circle notations with a C inside of them that decide which transition to be taken depending on the condition that is specified.

It is important to remember that attack trees do not differentiate between events and states. That means a tree node can either refer to an event or state occurrence. Consider as an example an attack tree for a shopping cart system with a *Sign In* node. This node can either refer to an event, where the intended meaning is that the attacker requests to go to the sign in page. The node can also refer to a state, then what is intended is for the user to complete the steps required to successfully sign in. On the other hand, statechart has a clear semantic difference between what is considered to be a state or an event. In Section 5 we will discuss the integration process, where step 4 checks if a leaf node is an event and then determines its equivalent transition. Steps 3 and 5 of Section 5 handle tree nodes that should be represented in the behavioral model as states. In Ref. [7] we discuss conditional entrances and give a more detailed discussion of how to handle events and states.

In addition to states, transitions, events and conditional entrances there is a need to depict the decision making process of an attacker and make it part of the behavioral model. The decision making process in attack trees is composed of a logical gate and its input. The gate type reflects the kind of decision the attacker is making. Gate inputs reflect the option given or steps needed for an attacker to accomplish a certain goal. Attack trees make use of two types of gates to depict the attacker's decision: conjunctive (AND) and disjunctive (OR) gates. AND gates are used to represent the set of actions that should be done together in order to achieve the parent node. OR gates are used to represent the set of different options, where performing at least one of these options will achieve the parent node. In addition to these two gates, a statechart representation for a Priority AND gate will be introduced. The difference between an AND gate and a Priority AND gate is that in an AND gate there is no consideration for the order of input occurrences. Therefore, a total of three types of gates will be given statechart representations.

The following subsections describe how to transform the three gates: AND, OR, and Priority AND into statechart notations. The statechart representations used here

Copyright by Institute of Software, Chinese Academy of Sciences. All rights reserved

to represent the gates and the system behavior are based on Harel's work[31]. The main focus is to introduce conversion rules that convert a gate with its inputs into statechart notations. The representation for each gate follows a modular approach. This will allow flexibility, component reuse and augmentation in the threat representation. Figure 2 shows a general statechart notation of a gate.

It can be seen from Fig. 2 that the statechart representation for any gate is composed of several orthogonal components. One component depicts the behavior of the gate itself while the rest of the components depict the gate inputs. In the gate component, the *Initial State* is the state before the occurrence of the gate and any of its inputs. On the other hand, the *Gate occurs* state is reached when the criteria for the gate inputs are met. The *Gate Representation* state is a super-state, represented in the figure as a dotted state, which represents the gate and its interaction with its inputs. The next subsections will describe the specific notation for this super-state for every gate that is going to be represented.

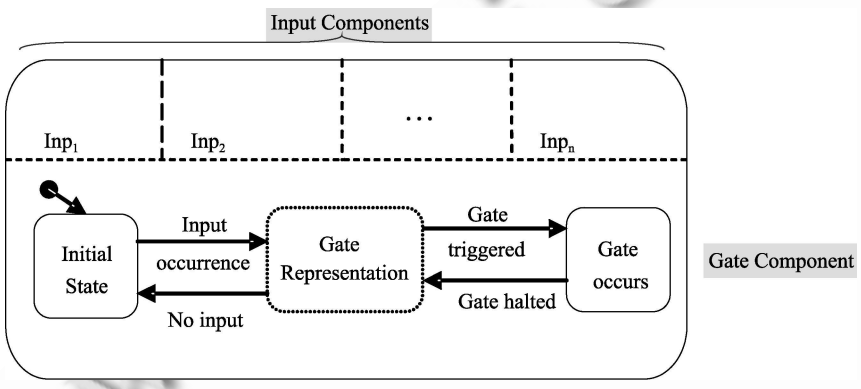

Figure 2. Statechart representation for a general gate

In an attack tree, a node (gate input) represents a step or a condition that should be met by an attacker. This node can either be an event, a state or a conditional statement. Gate inputs will be represented in statecharts as orthogonal components only when these nodes do not directly map into events. In case an attack tree node stands for an event, then the node can be directly represented by the gate component. The statechart representation for the gate component can either follow a monotonic behavior or not, depending on the particularity of the system that is being designed. In other words, a gate can either be modeled so that when a step is met it cannot be done (monotonic) or it can be undone. For example, Fig. 2 shows that when a gate is triggered through the occurrence of a *Gate triggered* event, it can step occurring when a *Gate halted* event is triggered. In the rest of this section, we will model the gate components such that when a gate is triggered (all its inputs are met) it cannot be undone.

The behavior of an attack is therefore modeled through the interaction of its components with each other and with the environment. Components interact with each other through broadcasting of events and through the use of actions. Not all of the components that comprise the security attack can be considered as self-contained. Input components are self-contained entities and can be reused, while gate components are not. This is because the behavior of gates depends on the behavior of their inputs.

Copyright by Institute of Software, Chinese Academy of Sciences. All rights reserved

This is expected because an attacker's capability of decision making (the presence of a gate) is dependent on the options (gate inputs) on hand.

4.1 Rule 1 - AND gate (conjunctive refinement)

An AND gate is represented using orthogonal states. The statechart notation for an AND gate can be seen as two parts. The first part represents the gate inputs while the second part represents the AND gate. It includes an orthogonal component for every gate input. The second part (the AND component) is a component that combines and manages all the standalone inputs. This AND component shows the interaction of the inputs together. Through the use of orthogonality, the AND gate representation will not modify or add semantic ambiguities to the existing statechart components. This representation has a general form irrespective to the number of inputs.

Figure 3 shows an example of an AND gate with 2 inputs A and B. It can be seen that for every gate input an orthogonal state is formed. The *Incr* event indicates the transition to a state where one of the child conditions (either A or B) is now present. While a *Decr* event indicates the loss of availability of one of these inputs. Consequently, there is a need for two consecutive *Incr* events in order for an AND gate of two inputs to be triggered.

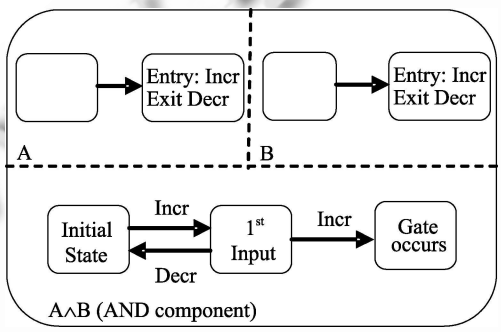

Figure 3. AND gate representation using orthogonality

4.2 Rule 2 - Priority AND gate (conjunctive refinement with order consideration)

The statechart representation of a Priority AND gate is a special case of the AND gate statechart representation. The only difference is in the Priority AND component, where the order of events should be taken into consideration. The priority AND gate occurs if all the gate inputs happen in a specific order. Therefore, the statechart representation of this gate should keep track of the targeted sequence of gate inputs. When all the inputs stand for events, the priority AND is represented as a set of consecutive states and transitions where (1) The number of states is less than the number of gate inputs by one. (2) The number of transitions is equal to the number of inputs. (3) The transitions should follow the same order of the gate inputs.

An example is a priority AND gate with three inputs A, B and C where the input sequence is C followed by A finally followed by B. Figure 4 shows the representation of this gate where orthogonal components are used to trigger the gate inputs. In case where an input is an event, then there is no need for its respective orthogonal state.

Copyright by Institute of Software, Chinese Academy of Sciences. All rights reserved

Another thing to notice in this representation of a priority AND gate is the assumption of monotonicity. Once a step is accomplished in the *Priority AND component*, it cannot be undone. In case the actions of an attacker should be undone then the representation can be modified by adding exit actions for each orthogonal components and adding transitions to the *Priority AND component*.

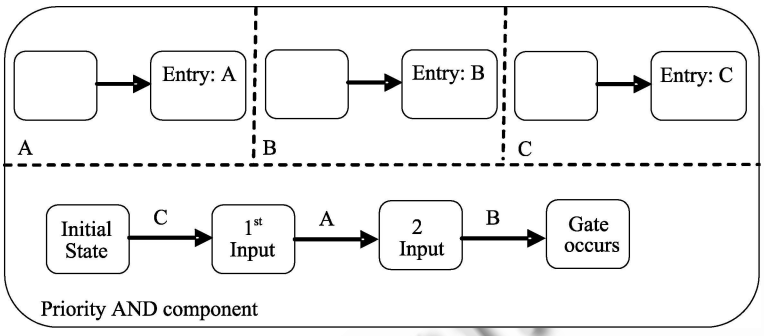

Figure 4. Statechart notation representing a priority AND gate

4.3 Rule 3 - OR gate (disjunctive refinement)

An OR gate is transformed into a set of transitions (one transition for each input) from the *Initial State* to the *Gate occurs* state. For an OR gate of two inputs A and B, the transition to *Gate occurs* state happens if either A, B, or both A and B occur. In the statechart representation, the occurrence of both A and B together is not represented because no additional information is added and will only complicate the representation. Figure 5 shows the statechart representation of an OR gate of two event inputs. In case the gate inputs cannot be represented by the system statechart as events or transitions, then the only difference in the OR gate is that an additional orthogonal component will be added for each gate input. In this case, gate inputs will be triggered by these orthogonal components.

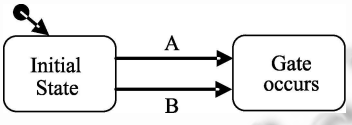

Figure 5. OR gate representation

5 The Integration Process

Our approach integrates security threats, which are represented in the form of attack trees, into functional specifications. The process starts by integrating one attack tree into the statechart of the system, where vulnerabilities in the system are identified and then mitigated. The process continues with another attack and repeats again until the rest of the attacks are integrated, analyzed and the threats are mitigated. The result will be the formation of a security model, where the model incorporates both system specifications and security threats. The method uses a set of systematic integration steps that concentrates on gates, where each gate with its

Copyright by Institute of Software, Chinese Academy of Sciences. All rights reserved

child conditions is considered on its own and integrated into the system statechart. The statechart representation of the security attack will then be composed of a set of orthogonal gate components. The statechart component of the root node will keep track of the occurrence of all the child conditions (sub-goals) and the attacker's ultimate goal (the root node). The rest of the orthogonal statechart components will represent individual sub-goals.

The proposed integration process consists of eight steps for transforming and integrating one attack tree into a system statechart. In step one, a threat formula (the semantics of an attack tree) is deduced. Steps two to five deal with the mismatches between the two heterogeneous models by gathering additional information which will be used by step six. In step three, the syntactical differences between the two models is considered. Steps two, four and five on the other hand deal with the semantic differences of the two models. Step six uses the information derived by the previous steps in order to merge the security attack with the system specification. The result is an integrated model that depicts the behavior of the system during an attack. From this model and through the verification of the system functionalities (step seven), vulnerabilities in the design are identified. The last step is to modify the integrated model by mitigating the system vulnerabilities. These mitigations or corrections in the behavioral model will result in a secure model. Each step will be discussed in details below:

Step 1 – ***Deduce a threat formula (a Boolean expression formula) from the attack tree***

The semantics of an attack tree is deduced using a formula that only shows the root node and how this root node can be reached through the combination of gate notations and leaf nodes. The BNF notation for the threat formula is shown below:

<**Threat-Formula**> ::= <Gate>
<**Gate**> ::= "("<operator> ", " <Gate-Inputs>")"
<**Gate-Inputs**> ::= <leaf-node> ", " <leaf-node> | <leaf-node> ", " <Gate> | <leaf-node> ", " <Gate-Inputs> | <Gate> ", " <Gate-Inputs>
<**operator**> ::=∧ | ∨

Figure 6. BNF notation for the threat formula

Step 2 – ***Check for any availability of Priority AND gates***

Attack trees have only two types of gate notations (refinements): OR and AND gates. Therefore, there is no special gate notation to represent child conditions that should be performed in a certain order. Instead, regular AND gates are used where the gate inputs (child conditions) are ordered accordingly. This ambiguity, whether there is a need to consider order or not, is only applicable to an AND gate. To resolve this type of ambiguity, this step checks every AND gate in the threat formula and replaces it with a Priority AND gate when order of occurrence is needed.

Step 3 – ***Decompose leaf nodes into simple definitions***

The leaf nodes of an attack tree are defined during the threat modeling process[5]. This process starts with the identification of the assets that should be protected. Next, the system architecture is constructed and then decomposed in order to derive a security profile. Finally the threats are identified, represented as attack trees and

Copyright by Institute of Software, Chinese Academy of Sciences. All rights reserved

then ranked. Therefore, the leaf nodes of an attack tree are highly dependent on the defined security profile and system architecture. That means the scope of the threat analysis determines which system components and threats to be considered as leaf nodes. On the other hand, a statechart diagram focuses on the functional behavior of a system. As the two models have different scopes and different level of abstractions, the level of detail for an event or a system component in a statechart and an attack tree representation might be different.

To resolve the ambiguities and differences in scope between the two models, we will use the term "simple definition" when a leaf node can be expressed by one of the following statechart notations: a conditional statement, a state occurrence, an event or a transition. This is shown below by the BNF notation:

<**leaf-node**> ::=<simple-definition> | <composite-definition>
<**composite-definition**> ::=<operator> ", " <simple-definition> ", "<simple-definition> |
<composite-definition> ", " <simple-definition>
<**simple-definition**> ::= "conditional statement" | "state occurrence" | "event" | "transition"
<**operator**> ::=$\wedge$ | $\overline{\wedge}$

Figure 7. BNF notation for a leaf node

When a leaf node in an attack tree has the same level of detail as a statechart notation that means it can be expressed as a simple definition, otherwise it is a composite definition and is composed of more than one simple definition. In order to continue with and at the same time simplify the integration process, we need the threat formula to be composed of simple definitions. This is done by checking every leaf node if it is a simple definition or not. If the leaf node does not have a simple definition, then this step decomposes it into simple definitions. The decomposition will represent the leaf node as an AND gate (or a Priority AND gate if the order of occurrence is important) whose inputs are simple definitions. An example is a leaf node A that occurs when both events B and C occur. In this case the leaf node should be decomposed into:

`A = (`$\wedge$`, B, C)` where the order of occurrence is not important
`A = (`$\overline{\wedge}$`, B, C)` where the order of occurrence is important
Finally the threat formula (deduced in step 1) is updated to reflect the changes.

Step 4 – ***Deduce the semantics table:***

Not all the activities or leaf nodes in an attack tree can be represented by the functionalities of the system. Some of the leaf nodes might refer to hardware components and cannot be represented by the specifications of the software. Other leaf nodes, such as normal events that can be exploited in order to cause a threat, might already be represented in the statechart and representing them again can cause redundancy. Therefore, leaf nodes in an attack tree can belong to one of the following groups:

- **Intrinsic Behavior** (Group 1)**:** Leaf nodes that belong to the behavior of the system, and can be directly represented by the software statechart.
- **External Behavior** (Group 2)**:** Leaf nodes whose functional behavior belongs

Copyright by Institute of Software, Chinese Academy of Sciences. All rights reserved

to other software applications that the current system depends on (such as the operating system, and the browser).

- **Malicious Behavior** (Group 3)**:** Leaf nodes that refer to hardware components, external system environment, or special software used by the attacker for malicious purposes. These leaf nodes cannot be represented by the system functionalities or by software applications that the current system depends on.

This step helps in identifying whether the simple definitions in a threat formula cannot be represented, are already represented, or can be represented by the system statechart. Here, functionalities are associated on a component basis. Different background colors will be used to distinguish statechart components of different group types. White, gray, and dark gray colors will identify statechart components of Intrinsic, External, and Malicious behavior respectively. This differentiation will be helpful to better understand the attack. It is enough to quickly look at the statechart representation of an attack and easily identify the components that are part of the system behavior and the components that are not. The need for a security specialist might be required to clarify semantic mismatches such as which statechart component matches the component responsible for the leaf node, or which software application and its respective component is responsible for the behavior of the leaf node.

The semantics table is a table that shows for every simple definition in the threat formula (from step 3) the equivalent statechart interpretation, whether it is part of the system behavior or not. If no interpretation was found, then this simple definition has a level of abstraction that is different from the functional model. The main emphasis in statecharts is on states while in an attack tree the emphasis is on events and their occurrences. Therefore, the focus when building the semantics table will be on transitions. Each row has four fields as shown in Fig. 8: a simple definition, the leaf node group, the statechart component, and equivalent transitions.

<Semantic-Table> ::= <simple-definition> <leaf-node-group> <statechart-component> <Transitions>
<leaf-node-group> ::= “Group 1”| “Group 2”| “Group 3”
<statechart-component> ::= “None”| <component> | <sub-component>
<Transitions> ::= “None”| {<transition>}
<transition> ::= <state-occurrence> [<state-nonoccurrence>]

Figure 8. BNF notation for the semantic table

An example for an exact interpretation: a “sign in” leaf node is exactly the same as the occurrence of a “login” in the component that is responsible for signing in. An example for similarity in interpretation: an “open HTML source code” leaf node is similar to the occurrence of “view HTML source code”, where open might indicate the ability to edit the code while view does not but both display the source code.

Attack trees do not differentiate between events and states and therefore cannot represent state persistence. It is important when depicting the behavior of an attack to not only represent the transition that causes the occurrence of a certain state, but also to include the transition that causes the departure from this state. In other words, it is important to model both the occurrence and nonoccurrence of a gate

Copyright by Institute of Software, Chinese Academy of Sciences. All rights reserved

input (leaf node). Representing a leaf node in the semantics table as only a transition from one state to another due to the occurrence of the event is not enough. There is a need for another transition that represents the leaf node when it becomes false (the change in input occurrence from being true to being false). This transition is indicated in Fig. 8 as "state-nonoccurrence".

Step 5 – ***Expand primitive functional and threat conditions (leaf nodes)***

This step expands the security formula by working on all its leaf nodes so that semantic mismatches are resolved. After being done with this step, both the threat formula and the semantic table are updated to reflect the recent changes. Although step 3 made sure that all the leaf nodes are simple definitions and can be represented using statechart notations, there might still be semantic differences between attack trees and the system model. These differences are the result of mismatches in the level of abstraction between some of the gate inputs. There are two cases that cause that:

- The behavior of the leaf node cannot be directly reached from the current state of the system. In this case, there are missing transitions that should be added. This type of mismatch is resolved by adding missing transitions as additional child conditions directly to the gate that contains the leaf node.

- The leaf node matches an event while what is intended is instead a sequence of events. In other words, the desired meaning is to successfully achieve and complete some desired set of actions that starts with or includes the leaf node. For example a Login leaf node might be interpreted as an event that allows the user to enter the login information, while the intended meaning is to successfully login to the system. This type of mismatch is resolved by replacing the leaf node by a sequence of transitions that modify the behavior of the system from its current state to the state that is expected to be achieved by the leaf node. This is done by replacing the leaf node by an AND gate (or a Priority AND gate if the order of occurrence is important) whose gate inputs are the set of transitions. It is sometimes possible to achieve the end (desired) state in more than one way. In that case, an OR gate is used and its gate inputs are the possible ways to achieve the same result.

The step is usually applied to the leaf nodes that map to either the functional behavior or the threat behavior. After passing through and expanding all the leaf nodes that have semantic mismatches, both the threat formula and the semantic table are updated to reflect the recent changes.

Step 6 – ***Transformation of gates in the threat formula***

This step transforms the threat formula into statechart representation and then merges the threat with the system specification. Gate transformation is done by working on the formula from left to right starting with the first gate then recursively working on the child conditions (inputs) that are composed of further refinements (gates). For each gate, we apply the respective transformation rule to depict the gate as statechart notations. If the gate inputs (simple definitions) have an equivalent representation in the semantics table, then use this equivalent representation during the construction of the gate's statechart notation. The statechart representation

Copyright by Institute of Software, Chinese Academy of Sciences. All rights reserved

for the first gate in the threat formula will be referenced as the threat controller component.

After all of the gates are transformed into statechart notations, the new representations of the attack tree are added as orthogonal components to the original statechart. The last thing to do is to modify the system behavior that has been targeted in order to indicate a threat occurrance. This is done by adding a new state. This state will be triggered by the action event that is produced by the threat controller component.

Step 7 – ***Verification – Identification of Vulnerabilities***

The integration of a threat model (an attack tree representation in statechart notations) into the system functionalities allows either the verification of threat absence (the attack cannot be achieved) or threat presence (the attack can be completed successfully) in the design. The verification process is a fundamental step in proving that the specifications of a system have no ambiguities, errors, conflicts or inconsistencies. The aim of our approach is to demonstrate that integrating the behavior of security attacks, which is a nonfunctional behavior, into the system model has an important role in verifying the correctness of functional specifications. The integration process identifies the system functionalities and behavior that are involved in this specific attack. The integrated model not only identifies the system functionalities used during the attack but also identifies which of these functionalities have been used normally and which functionalities have been exploited.

The threat controller component describes the steps needed by an attacker to complete an attack. The behavior of an attack depends on functionalities that are correct and allowed by the system and on illegal behavior that is not part of the system functionalities but still allowed by the behavior of the system. A system has a threat presence if the behavior of the system does not recognize an illegal action and considers it as part of the correct behavior. The presence of a threat indicates that the system design, and therefore the system, has a vulnerability that can be exploited by the user to successfully complete an attack.

Step 8 – ***Mitigation of system vulnerabilities***

The purpose for integrating attack trees into the functional model is to come up with a secure model. This model takes into consideration both insecure conditions and security features. Insecure conditions in this model are the attack tree components that are added to the system behavior by Step 6. Security features are responsible for how the system responds to insecure conditions by not allowing the attacker's ultimate goal to be achieved. That means security features are the corrections done to the functional model so that the vulnerabilities in the system are eliminated.

The integration of a threat model into the system functionalities allows either the verification of threat absence or threat presence in the system. In the case of threat presence, vulnerabilities in the system behavior are identified. The vulnerabilities found by the verification step are the starting point for correcting the behavior of the system and improving its quality. Each vulnerability should be mitigated by using security features. The choice of mitigation is usually influenced by the cost of eliminating the vulnerability and the purpose of the system.

Copyright by Institute of Software, Chinese Academy of Sciences. All rights reserved

6 Case Study

In this section we describe the application of the proposed approach to a shopping cart system. We start this section by briefly describing the application on hand and the security issues to consider. Next we demonstrate the complete process of integrating two attack trees into the shopping cart functional model, verifying the presence or absence of threat vulnerabilities and mitigating the identified vulnerabilities. We finally conclude this section through a summary of the lessons learned from this case study.

The shopping cart system allows a user to browse, select and purchase items online. The functional model of the shopping cart system is represented in Fig. 8. The system is composed of the following components:

- A shopping cart component that keeps track of the items that a customer is interested in purchasing. The component allows the customer to add and remove items from the current shopping session.
- A browsing component that allows a customer to browse the available items provided by the e-commerce seller and display detailed description for individual products.
- A sign in component that allows a customer to either create and login into a new account, or to login to an existing account.
- A checkout component that allows a customer to purchase the items that were added into the shopping cart.
- A back-end database that stores the customer information, product description and inventory.

From a company's perspective, security concerns mainly focus on the prevention of attacks that target the company's assets. Security attacks at the application level usually exploit existing vulnerabilities in the system (such as Improper design) in order to take advantage of the assets that are of value. Therefore, the main purpose of software security is to better protect these assets. For the shopping cart system, the assets that are of value to the e-commerce company are: (1) customer information such as login information, address, and credit card information, (2) the company's reputation and (3) inventory.

The main focus of the case study is to stress that the inclusion of security threats into the behavior of the system has an important role in verifying and correcting vulnerabilities in the functional specifications. The process starts with integrating one attack tree into the statechart of the shopping cart system where vulnerabilities in the system are identified and then mitigated. The process continues with another attack and repeats again until the rest of the attacks are integrated, analyzed and the threats are mitigated. In this case study, we have transformed ten attack trees into statechart notations and integrated them into the functional model. In order to analyze the behavior of the system under different types of attacks and thus make the study more effective we made sure that the chosen attack trees cover all the different threat categories of Microsoft's STRIDE model[5]. More than twenty vulnerabilities

Copyright by Institute of Software, Chinese Academy of Sciences. All rights reserved

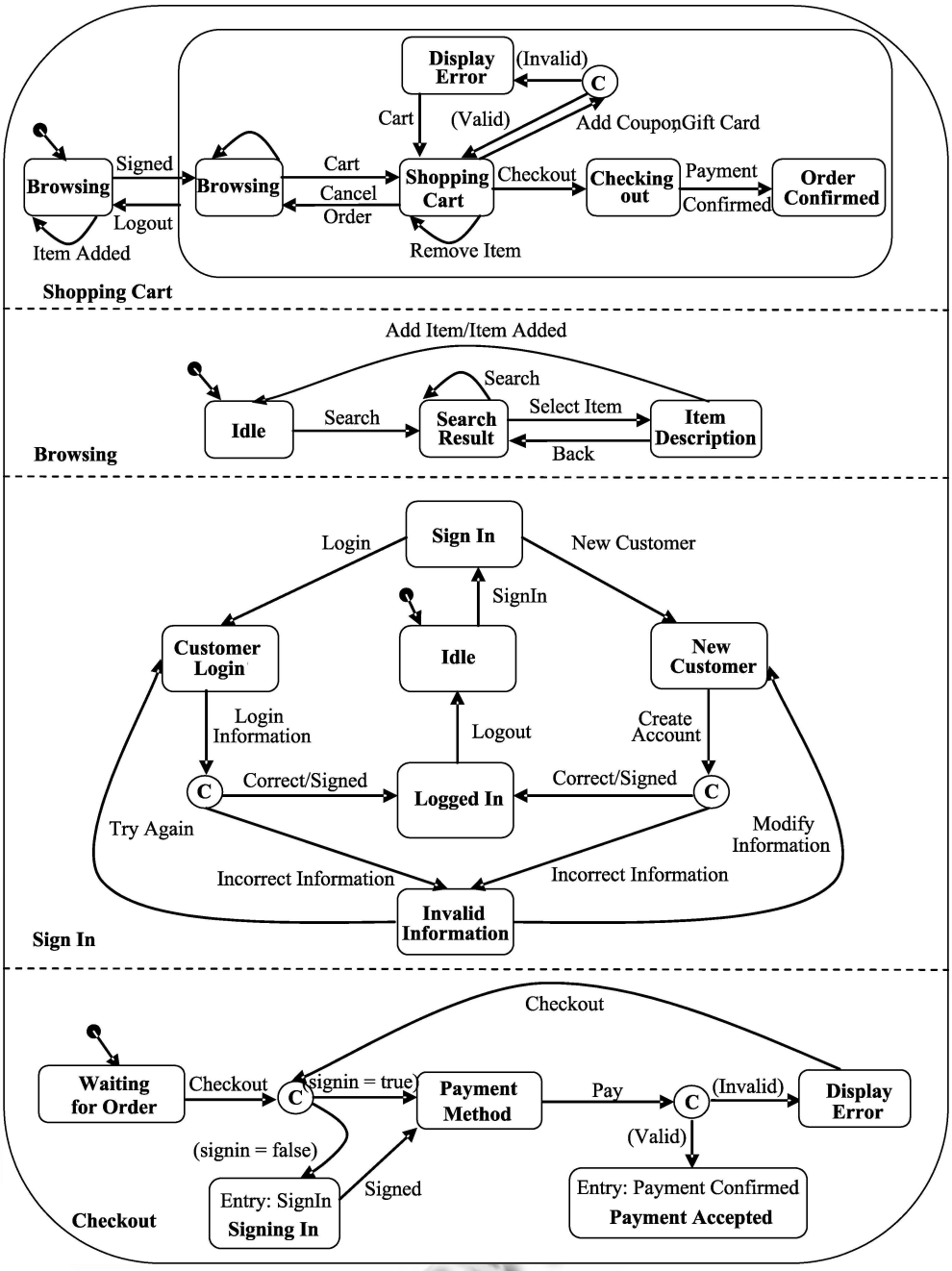

Figure 9. Statechart for a shopping cart system

in the system were identified and mitigated. As the shopping cart application is complex and depends on an untrustworthy computing environment (the Internet), the list of identified threats was by no means exhaustive. Nevertheless, it provided a fairly good basis for demonstrating the effectiveness and benefits of our work.

Copyright by Institute of Software, Chinese Academy of Sciences. All rights reserved

6.1 Example 1: Purchase an item at a reduced price

In this section we will demonstrate in details the integration process of the threat of purchasing an item at a reduced price into the shopping cart statechart. This attack will allow the malicious user to purchase an item at a price that is cheaper than the actual price. The attack tree of Fig. 10 shows the required steps. A malicious user first logs in into his/her user account and then modifies the price of a selected product, adds it to the shopping cart and proceeds with the checkout process.

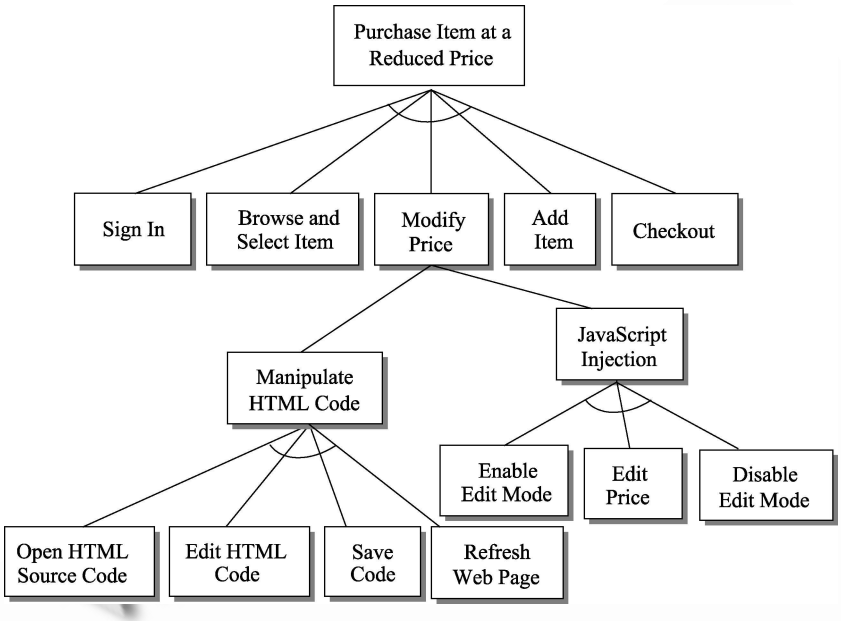

Figure 10. Attack tree for purchasing an item at a reduced price

There are two possible ways for a user to modify the price of an item: (1) Modify the price of an item in the HTML source code. Some shopping cart applications use hidden fields in the HTML source code to store the price and quantity of items that are in the shopping cart. This type of hidden information might be used maliciously by a customer to buy items at a very cheap price. This vulnerability was found and exploited in numerous online shopping cart applications[35]. (2) Inject JavaScript code in the address bar of the web browser in order to edit the price of the item.

Step 1 – ***Deduce a threat formula from the attack tree***

Here a successful completion of an attack (which is the root node) is done by purchasing an item at a price different from what is indicated in the web site. This attack will only occur through the combination of the gates and their leaf nodes as shown below:

Purchase Item at a Reduced Price = (∧, Sign In, Browse and Select Item, (∨, (∧, Open HTML Source Code, Edit HTML Code, Save Code, Refresh Web Page), (∧, Enable Edit Mode, Edit Price, Disable Edit Mode)), Add Item, Checkout)

As it is the case that attack trees do not differentiate, in the case of AND gates, between when the order of occurrence should be considered and when it should not, the next step makes this distinction an explicit one.

Step 2 – ***Check for any availability of Priority AND gates***

Copyright by Institute of Software, Chinese Academy of Sciences. All rights reserved

The threat formula has four gates. Three of these gates are AND gates where the order of occurrence of the inputs should be considered. Therefore the three AND gates should be replaced by a Priority AND gate. The threat formula is now the following:

Purchase Item at a Reduced Price = (⩓, Sign In, Browse and Select Item, (∨, (⩓, Open HTML Source Code, Edit HTML Code, Save Code, Refresh Web Page),(⩓, Enable Edit Mode, Edit Price, Disable Edit Mode)), Add Item, Checkout)

The next thing to consider is the mismatches between the attack tree and the functional behavior of the system. We start with differences in syntax by going through each leaf node and checking if it can be directly represented using statechart notations.

Step 3 – ***Decompose leaf nodes into simple definitions***

For every leaf node we check if it is one of the following: an event, a transition, a state occurrence, or a conditional statement. If this is the case, then the leaf node is a simple definition and we leave the leaf node as it is. If not, then the leaf node is a composite definition and should be decomposed further. Here, all of the leaf nodes in the threat formula are simple definitions except for the *Browse and Select Item* leaf node. This leaf node is made up of two events, a browsing event followed by selecting an item. As the order of event occurrence is important, then the leaf node will decomposed into a Priority AND gate of two inputs:

Browse and Select Item = (⩓, Browse, Select Item)

The threat formula now becomes:

Purchase Item at a Reduced Price = (⩓, Sign In, (⩓, Browse, Select Item),(∨, (⩓, Open HTML Source Code, Edit HTML Code, Save Code, Refresh Web Page), (⩓, Enable Edit Mode, Edit Price, Disable Edit Mode)), Add Item, Checkout)

The next thing to do is check for semantic differences.

Step 4 – ***Deduce the semantics table***

This step helps in further defining the behavior of each simple definition. First, the simple definition is mapped to the group that represents its functionality. Then the statechart component and the transitions needed to achieve the behavior of the simple definition are added to the semantics table. If the behavior of the simple definition is part of Group 1, then the semantics table indicates whether it can be represented or is already represented by the system statechart. The semantics table for the threat formula is shown below in Table 1. From this semantic table we can see which of the twelve simple definitions belong to the system behavior (in this case five belong to Group 1) and their respective transitions.

Table 1 The semantics table

Simple Definition	Group	Statechart Component	Equivalent Transition
Sign In	Group1	Sign In	(SignIn.Idle, SignIn, SignIn.SignIn)
Browse	Group1	Browsing	(Browsing.Idle, Search, Browsing.SearchResult)
Select Item	Group1	Browsing	(Browsing.SearchResult, Select Item, Browsing.ItemDescription)
Open HTML Source Code	Group2	Web Browser-View Menu	(WebBrowser.ViewMenu, Source, Editor.SourceCode)
Edit HTML Code	Group3	Editor	None
Save Code	Group3	Editor	(Editor.SourceCode, Save, Editor.SourceCode)
Refresh Web Page	Group2	Web Browser	(WebBrowser.Main, Refresh, WebBrowser.Main)
Enable Edit Mode	Group2	Web Browser	(WebBrowser.Main, JSEnableEditing, WebBrowser.Main)
Edit Price	Group2	Web Browser	(WebBrowser.Main, JSEditPrice, WebBrowser.Main)
Disable Edit Mode	Group2	Web Browser	(WebBrowser.Main, JSDisableEditing, WebBrowser.Main)
Add Item	Group1	Browsing	(Browsing.ItemDescription, Add Item, Browsing.Idle)
Checkout	Group1	Shopping Cart	(ShoppingCart.ShoppingCart, Checkout, ShoppingCart.CheckingOut)

Copyright by Institute of Software, Chinese Academy of Sciences. All rights reserved

Step 5 – ***Expand primitive functional and threat conditions***

This step is concerned with the semantic differences that are the result of mismatches in the level of abstraction between the gate inputs. There is one leaf node that matches the case where the behavior of the simple definition cannot be directly reached from the current state of the system: *Checkout*. The current state of the shopping cart system (as can be seen from the transitions in the semantics table) is the browsing state where the user has already signed in. That means in order to checkout the user should be in the Shopping Cart state rather than in the browsing state. What is missing is a *cart* transition that should occur after the *Add Item* transition but before the *Checkout* transition. The threat formula is now expanded with the missing transition as shown below:

Purchase Item at a Reduced Price = (⊼, Sign In, (⊼, Browse, Select Item),(∨, (⊼, Open HTML Source Code, Edit HTML Code, Save Code, Refresh Web Page), ⊼, Enable Edit Mode, Edit Price, Disable Edit Mode)), Add Item, Cart, Checkout)

Simple definitions that should match a sequence of events instead of a single event should also be expanded. There are three cases in the threat formula:

- Sign In: refers to an event that allows the user to enter the login information, while the intended meaning in the attack tree is to successfully login to the system. This behavior can be achieved in two ways, either by creating a new account or by logging in to an existing account. The expanded behavior is:

  Signed In = (∨, (⊼, SignIn, Login, Login Information, Signed), (⊼, SignIn, New Customer, Create Account, Signed))

- Open HTML Source Code: in order to open the HTML source code, the user should first click on the browser's view menu. The expanded behavior is:

  Open HTML Source Code = (⊼, view, Source)

- Checkout: what is intended is that the user successfully completes the checkout process rather than is just starting with the checkout process. The expanded behavior is:

  Checked out = (⊼, Checkout, Pay, Payment Confirmed)

Both the threat formula and the semantic tree are modified to reflect the new changes. Therefore, the threat formula is now the following:

Purchase Item at a Reduced Price = (⊼,∨, (⊼, SignIn, Login, Login Information, Signed), (⊼, SignIn, New Customer, Create Account, Signed)), (⊼, Browse, Select Item), (∨, (⊼, (⊼, view, Source), Edit HTML Code, Save Code, Refresh Web Page),(⊼, Enable Edit Mode, Edit Price, Disable Edit Mode)), Add Item, Cart, (⊼, Checkout, Pay, Payment Confirmed))

Step 6 – ***Transformation of gates in the threat formula***

In this step we start transforming the threat formula into statechart representation by first working on the first gate:

Purchase Item at a Reduced Price = (⊼, Signed In, Browse & Select Item, Modify Price, Add Item, Cart, Checked out)

Where:

Signed In = (∨, (⊼, SignIn, Login, Login Information, Signed), (⊼, SignIn, New Customer, Create Account, Signed))

Copyright by Institute of Software, Chinese Academy of Sciences. All rights reserved

Browse & Select Item = ($\overline{\wedge}$, Browse, Select Item)

Modify Price = ($\vee$, ($\overline{\wedge}$, ($\overline{\wedge}$, view, Source), Edit HTML Code, Save Code, Refresh Web Page), ($\overline{\wedge}$,Enable Edit Mode, Edit Price, Disable Edit Mode))

Checked out = ($\overline{\wedge}$, Checkout, Pay, Payment Confirmed)

The first operator is a Priority AND gate of six child conditions, where two of the gate inputs are event while the rest are sub-goals. Figure 11 shows a partial statechart representation of the gate where the inputs that are not events are represented as orthogonal states. The statechart representation of both *Add Item* and *Cart* is based on their respective transitions in the semantics table, and the events that cause the occurrence of these simple definitions are directly represented in the threat controller component. The occurrence of the orthogonal states will trigger an action that causes the controller component to change its current state.

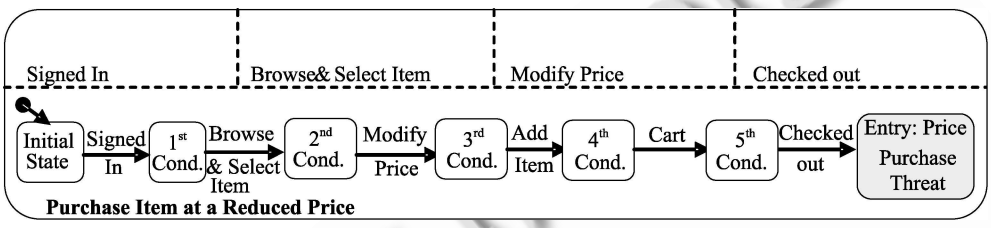

Figure 11. Statechart notation representing the threat controller component

The next step is to represent the four orthogonal states starting with the *Sign In* component:

Signed In = ($\vee$, ($\overline{\wedge}$, SignIn, Login, Login Information, Signed), ($\overline{\wedge}$, SignIn, New Customer, Create Account, Signed))

Signed In = ($\vee$,Current User, New User)

Signed In is an OR gate of two inputs, where each input is a Priority AND gate of four inputs. Figure 12 shows a partial statechart representation for this gate using the transformation rule for the OR gate.

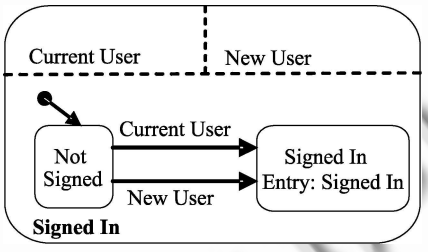

Figure 12. Statechart notation representing the Signed In component

The next step is to give statechart notations to the two inputs of the OR gate. Starting with *Current User*:

Current User = ($\overline{\wedge}$, SignIn, Login, Login Information, Signed)

New User = ($\overline{\wedge}$, SignIn, New Customer, Create Account, Signed)

All of the inputs for this gate are simple definitions; there is no need for orthogonal states. Following the conversion rule for the Priority AND gate will result in the following statechart representation:

Copyright by Institute of Software, Chinese Academy of Sciences. All rights reserved

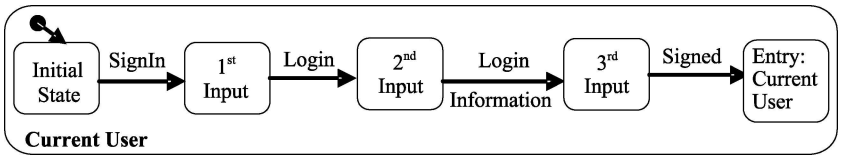

Figure 13. Statechart notation representing the Current User component

The *New User* Priority AND gate is very similar to the previous Priority AND gate. Figure 14 shows the statechart representation of this component:

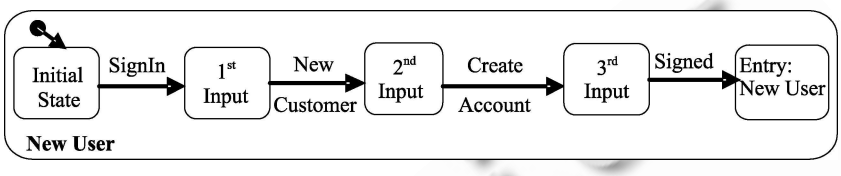

Figure 14. Statechart notation representing the New User component

The next step is to represent the *Browse & Select Item* component. It can be seen from the semantics table that both the Priority AND gate inputs are events. The semantics table also indicates that the *Browse* simple definition is represented by the system by the following transition: *(Browsing.Idle, Search, Browsing.SearchResult)*. That means it is enough to use the *Search* event in the statechart representation of the *Browse & Select Item* component. Figure 15 shows the representation for this component.

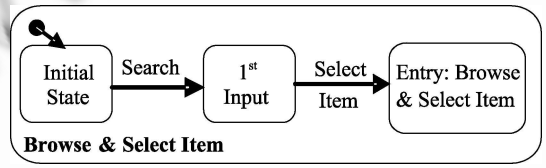

Figure 15. Statechart notation representing Browse & Select Item

As for the *Modify Price* component, this sub-goal is composed of an OR gate of two inputs, where both of tis inputs are also sub-goals as shown below:

Modify Price = (∨, Manipulate HTML Code, JavaScript Injection)

Where:

Manipulate HTML Code = (∧̄, Open HTML Source Code, Edit HTML Code, Save Code, Refresh Web Page)

Open HTML Source Code = (∧̄, view, Source)

JavaScript Injection= (∧̄, Enable Edit Mode, Edit Price, Disable Edit Mode)

Following the transformation rule for the OR gate, Fig. 16 shows the partial statechart representation for this gate. As can be seen from the figure below, the threat of modifying an item's price can either be done injecting JavaScript in the address bar of the web browser or by modifying the HTML source code. Different background colors are used to differentiate Group 2 and Group 3 from the functional behavior of the system (Group 1).

Copyright by Institute of Software, Chinese Academy of Sciences. All rights reserved

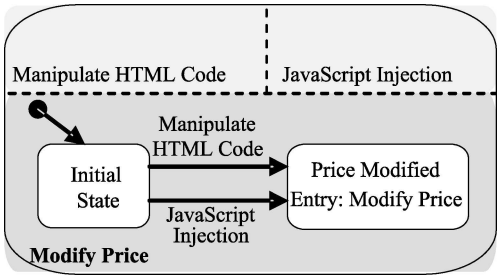

Figure 16. Statechart notation representing the Modify Price component

Next, the two sub-goals for the *Modify Price* component are given a statechart representation. *Manipulate HTML Code* is composed of a Priority AND gate of four inputs where the first input is a Priority AND gate of two inputs. Therefore, as shown in Fig. 17 the statechart representation has an orthogonal component that represents the composite input.

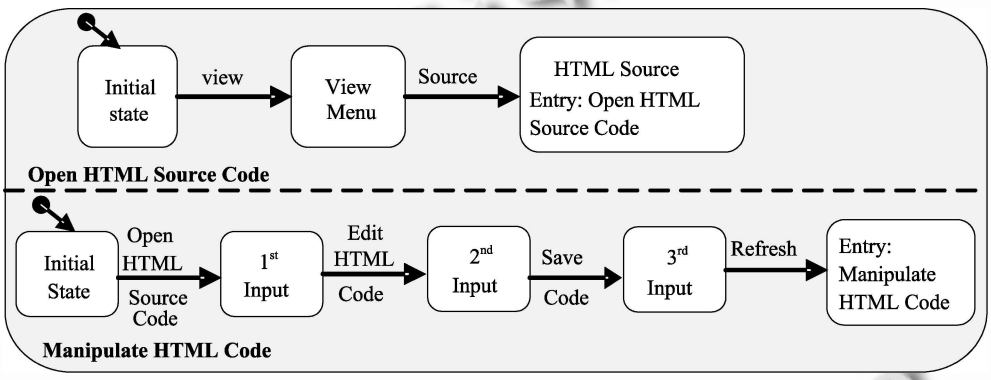

Figure 17. Statechart notation representing the Manipulate HTML code

As for the JavaScript Injection sub-goal, it is composed of a Priority AND gate of three inputs. The gate inputs are all simple definition. The statechart is shown below:

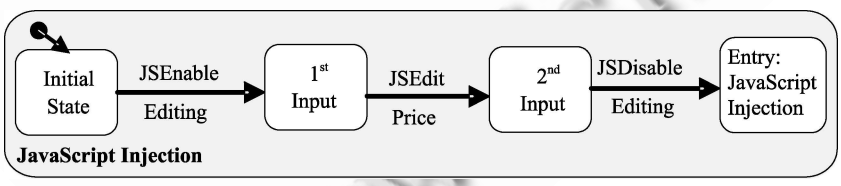

Figure 18. Statechart notation representing the JavaScript Injection

What is left is to represent the *Checkout* component, which is a Priority AND gate of three inputs. The gate inputs are all simple definition. The statechart representation is shown in Fig. 19.

The last thing to do is to integrate the newly constructed security threat components into the shopping cart behavioral model. First, the newly formed statechart

Copyright by Institute of Software, Chinese Academy of Sciences. All rights reserved

components are added to the system statechart as orthogonal parts. Next, the functional behavior of the system should be modified to indicate the occurrence of the attack. This is done by adding a new state to the *shopping cart* component. The transition to this state will be triggered when the yellow colored state in the threat controller component is reached and a *Purchase Threat* action is emitted. When this happens then the attacker's ultimate goal has been reached and the threats in the system have been exploited. Figure 20 shows the integrated statechart-based functional modeling with a security attack. The *Browsing*, *Sign In*, and *Checkout* functional components have not been modified during the integration process and thus are not shown by the figure in order to limit the size of the model.

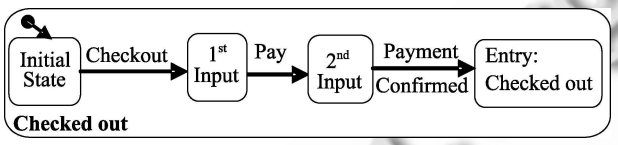

Figure 19. Statechart notation representing the Checked out component

Step 7 – ***Verification – Identification of Vulnerabilities***

Looking at the integrated model of Fig. 20, and in particular the threat controller component, shows that for the attack to occur six conditions should take place one after the other. Therefore the shopping cart system will have a security breach with the occurrence of the state with the yellow background color. It is straightforward, through the use of the integrated model, to identify the components or states that are responsible for triggering these six events.

The integrated model not only identifies the system functionalities used during the attack but also identifies which of these functionalities have been used normally and which functionalities have been exploited. All the threat components that have no background color are system functionalities. That means the *Signed In*, *Browse & Select Item*, *Checked out*, *Current User* and *New User* components from Fig. 20 belong to the functionalities in the system. Although the last component (*Purchase Item at a Reduced Price*) also has no background color, this statechart component represents the threat controller component.

From the integrated model we can see that there is one component that causes threat to the system: the *Modify Price* component (the Malicious Behavior component with dark gray background). To analyze which system functionalities are compromised, we first locate the event that triggers the *Modify Price* component. In the threat controller component in Fig. 20, the 3^{rd} *Cond.* state indicates that the price of the item has been modified locally at the client side but the changes are not yet reflected in the shopping cart system. After the occurrence of this threat (modifying the price), the system has a vulnerability if the behavior of the system does not recognize this threat and accepts it as a correct behavior. This is the case for the shopping cart system of Fig. 9, where the item is added with the modified price into the customer's shopping cart. This indicates that the system vulnerability is in the functionality of adding an item to the shopping cart. Similarly, for the attack to be completed successfully, the attacker should complete the checkout process and pay a reduced price for the purchased item. The checkout component in Fig. 9 does not

Copyright by Institute of Software, Chinese Academy of Sciences. All rights reserved

check whether the price of an item is valid or not. Therefore, the second vulnerability is the checkout process.

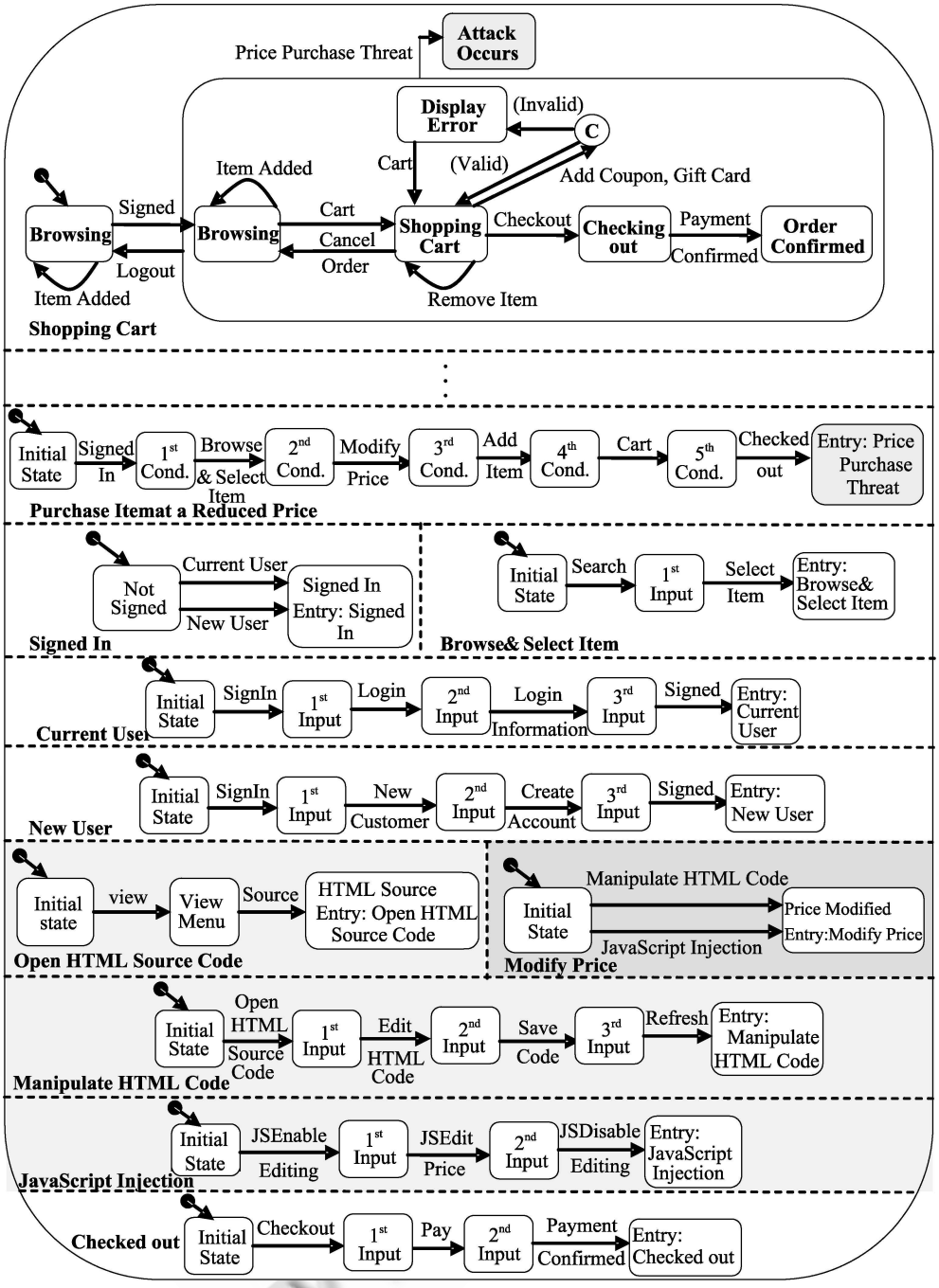

Figure 20. Integrated shopping cart model with threat concerns

Copyright by Institute of Software, Chinese Academy of Sciences. All rights reserved

Step 8 – ***Mitigation of system vulnerabilities***

The vulnerabilities found through the previous step are the starting point for correcting the behavior of the system and improving its quality. Two threats were identified by integrating the *Purchase Item at a Reduced Price* attack:

- In the *Browsing* functional component, the *Add Item* functionality allows the addition of an item without checking the values of the price and quantity fields. Therefore this threat, part of the *Modify Price* component of Fig. 20, belongs to Parameter *Manipulation* vulnerability, one of the application vulnerability categories[36].

- The *Checkout* functional component allows the item with the manipulated parameters to be purchased without making sure that the customer is paying the right price or not. This type of threat, part of the *Purchase Item at a Reduced Price* component of Fig. 20, belongs to *Input Validation* vulnerability, one of the application vulnerability categories.

There are two ways to prohibit the attack tree of Fig. 10 from occurring. (1) Either by preventing the customer to manipulate the data. The focus of the mitigation process will then be on the *Add Item* transition in the *Browsing* component. (2) The other way is by validating the price of the items before the completion of the checkout process. The key transition that the mitigation process should focus on will be the *Pay* transition in the *Checkout* component.

The first mitigation approach makes sure that when an item is placed in the shopping cart its price has not been modified. The second approach validates the price of each item in order to check if it is correct or not before proceeding with the payment process. That means it is enough to eliminate one of the identified vulnerabilities in the system in order to prevent the attack from occurring. In both of the mitigation approaches, the price of the items in the shopping cart should be validated. The first approach validates the price before adding the item into the shopping cart while the second approach delays the validation until the checkout process.

Here we consider the first approach for threat mitigation. In this case, the mitigation process can be one of the following ways:

Option1: validates whether the price of an item has been manipulated or not. The integrated shopping cart model should then indicate when a manipulation happens. This type of mitigation is useful for intrusion detection systems. One advantage for this type of mitigation is that the functional behavior does not have to be modified. The mitigation can be added as a separate statechart component. This is useful for separating security features from the functional model. Figure 21 shows the orthogonal component that validates the price of an item every time an *Add Item* transition occurs. If the price of the item from the client's browser is different from the store's price then the item will not be added to the shopping cart.

The *Shopping Cart* component should notify the user that the item cannot be added to the cart because the price is incorrect. Figure 22 shows the addition of a state that issues a notification to the user. This state will be triggered while the user is browsing and attempts to add an item with a manipulated price.

Copyright by Institute of Software, Chinese Academy of Sciences. All rights reserved

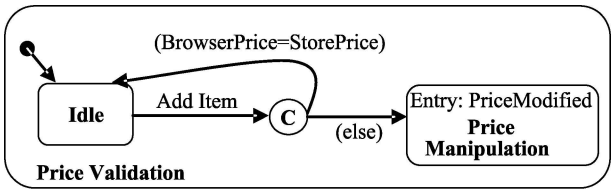

Figure 21. Price validation component

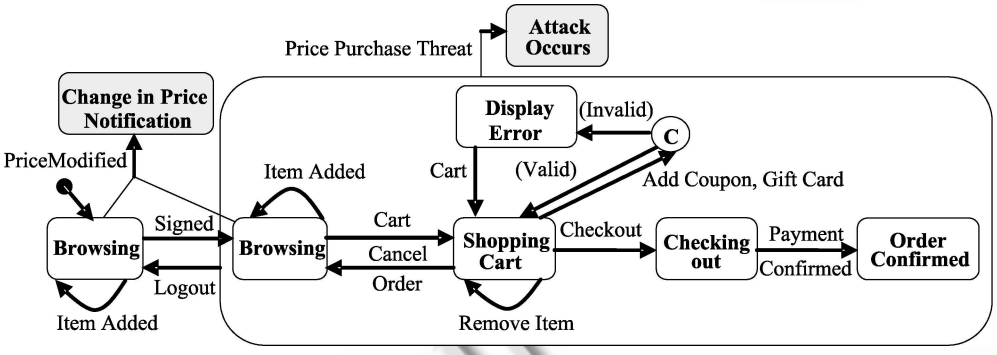

Figure 22. Modified shopping cart component

Option 2: the system identifies that the price of an item is manipulated. The integrated model then automatically corrects the behavior by adding into the shopping cart the item with the correct price rather than the manipulated one. This type of mitigation is useful for self healing software. In this case the *Browsing* component should be modified in order to mitigate the threat. Figure 23 shows the modified *Browsing* component. Every time a user requests an item to be added to the cart, the price is validated first. If the input was manipulated by the customer then the price of the item is changed to the correct value, this is shown in the figure as *Change Price* state, then the item is added to the cart.

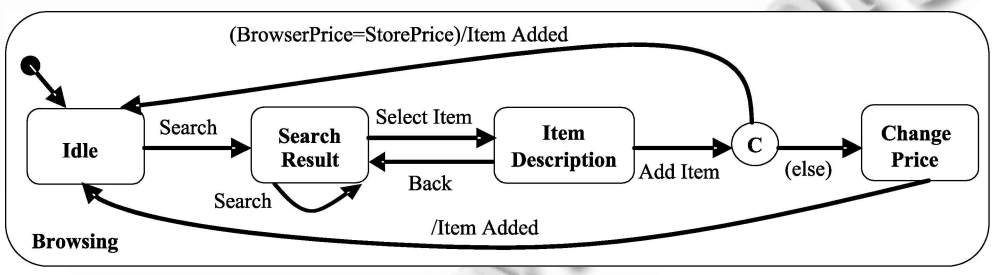

Figure 23. Modified browsing component with mitigation

6.2 Example 2: Coupon fraud

Another example of a security attack that targets a shopping cart application is of a coupon misuse. A coupon is usually introduced by an e-commerce seller to promote a special item or to encourage sale at a certain price. A coupon fraud occurs when the attacker applies a coupon on a product different from the intended one.

Copyright by Institute of Software, Chinese Academy of Sciences. All rights reserved

The usage of a coupon on a wrong item defies the coupon's purpose and can cause loss to the seller. The attack tree of a coupon fraud, represented in Fig. 24, shows the required steps needed for this type of an attack to be successful.

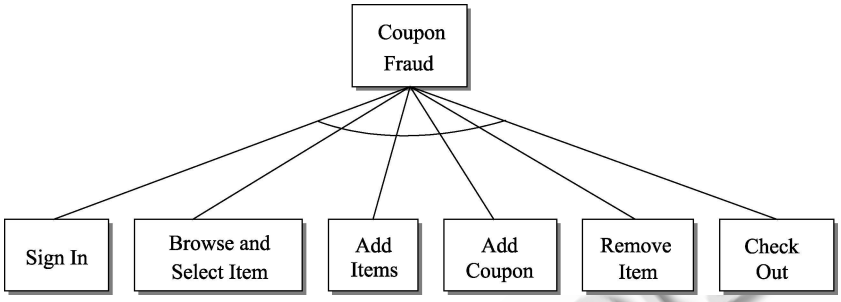

Figure 24. Attack tree for a coupon fraud

Here we describe the process of integrating this attack tree into the shopping cart statechart of Fig. 9. We start with the first step of the integration process

Step 1 – ***Deduce a threat formula from the attack tree:***

Successful completion of a fraud attack occurs when the root node of the tree in Fig. 10 is reached. This root node will only occur through the combination of the gates and their leaf nodes as shown below:

Coupon Fraud= (∧, Sign In, Browse and Select Item, Add Items, Add Coupon, Remove Item, Checkout)

Step 2 – ***Check for any availability of Priority AND gates:***

The coupon fraud threat formula has only one gate. This gate is an AND gate and the order of occurrence of its child conditions is important. Therefore this gate should be replaced by a Priority AND gate. The threat formula is now the following:

Coupon Fraud= ($\overline{\wedge}$ Sign In, Browse and Select Item, Add Items, Add Coupon, Remove Item, Checkout)

Step 3 – ***Decompose leaf nodes into simple definitions***

The next thing to consider is the mismatches between the attack tree and the functional behavior of the system. We start with differences in syntax by going through each leaf node and checking if it can be directly represented using statechart notations.Here, all of the leaf nodes in the threat formula are simple definitions except for the *Browse and Select Item* leaf node. This leaf node is composed of two events, a browsing event followed by a selecting event. In this case the order of event occurrence is important. Therefore, the leaf node will be decomposed into a Priority AND gate of two inputs:

Browse and Select Item = ($\overline{\wedge}$, Browse, Select Item)

The threat formula now becomes:

Coupon Fraud = ($\overline{\wedge}$, Sign In, ($\overline{\wedge}$, Browse, Select Item), Add Items, Add Coupon, Remove Item, Checkout)

Step 4 – ***Deduce the semantics table:***

The next thing to do is check for semantic differences, starting with the construction of the semantics table.This step helps in further defining the behavior of each simple definition. The semantics table for the threat formula is shown below in Table 2.

Copyright by Institute of Software, Chinese Academy of Sciences. All rights reserved

Table 2 The semantics table

Simple Definition	Group	Statechart Component	Equivalent Transition
Sign In	Group1	Sign In	(SignIn.Idle, SignIn, SignIn.SignIn)
Browse	Group1	Browsing	(Browsing.Idle, Search, Browsing.SearchResult)
Select Item	Group1	Browsing	(Browsing.SearchResult, Select Item, Browsing.ItemDescription)
Add Item	Group1	Browsing	(Browsing.ItemDescription, Add Item, Browsing.Idle)
Add Coupon	Group1	Shopping Cart	(ShoppingCart.ShoppingCart, Add Coupon, ShoppingCart.CheckingOut)
Remove Item	Group1	Shopping Cart	(ShoppingCart.ShoppingCart, Remove Item, ShoppingCart.CheckingOut)
Checkout	Group1	Shopping Cart	(ShoppingCart.ShoppingCart, Checkout, ShoppingCart.CheckingOut)

Step 5 – ***Expand primitive functional and threat conditions:***

In this attack there is only one instance where the behavior of the simple definition cannot be directly reached from the current state of the system: which is the *Add Coupon* leaf node. The current state of the shopping cart system (as can be seen from the transitions in the semantics table) is the browsing state where the user has already signed in and added an item. That means in order to add a coupon and complete the rest of the steps, the current state should be in the Shopping Cart state rather than in the browsing state. What is missing is a *cart* transition that should occur after the *Add Item* transition but before the *Add Coupon* transition. The threat formula is now expanded with the missing transition as shown below:

Coupon Fraud = (⊼, Sign In, (⊼, Browse, Select Item), Add Items, Cart, Add Coupon, Remove Item, Checkout)

Simple definitions that should match a sequence of events instead of a single event should also be expanded. There are three cases in this threat formula:

Sign In: refers to an event that allows the user to enter the login information, while the intended meaning in the attack tree is to successfully login to the system. This behavior can be achieved in two ways, either by creating a new account or by logging in to an existing account. The expanded behavior is:

Sign In = (∨, (⊼, SignIn, Login, Login Information, Signed), (⊼, SignIn, New Customer, Create Account, Signed))

Add Items: the customer here should add more than one product to the cart. One of these products should be the item that is part of the coupon promotion. Here the process of adding one item at a time is repetitive, and thus forms a cycle. The user first browse, selects, then adds an item. In case there is a need for more items, then the user will repeat the process of browsing, selecting and adding an item. Attack trees are incapable of representing cycles while statecharts on the other hand can easily depict repetitive behavior. After the first item is added to the cart the cycle is represented as a choice of either browsing again or performing the next child condition, in this case the *Cart* transition. The choice is depicted by using an OR gate. The modifications are shown below:

Coupon Fraud = (⊼, Sign In, (⊼, Browse, Select Item), Add Item, (∨, Browse, Cart), Add Coupon, Remove Item, Checkout)

Checkout: what is intended is that the user successfully completes the checkout process rather than is just starting with it. The expanded behavior is:

Checkout = (⊼, Checkout, Pay, Payment Confirmed)

Both the threat formula and the semantic table are modified to reflect the new changes. Therefore, the threat formula is now the following:

Coupon Fraud = (⊼,(∨, ⊼, SignIn, Login, Login Information, Signed), (⊼, SignIn,

Copyright by Institute of Software, Chinese Academy of Sciences. All rights reserved

New Customer, Create Account, Signed)), ($\overline{\wedge}$, Browse, Select Item), Add Item, ($\vee$, Browse, Cart), Add Coupon, Remove Item, ($\overline{\wedge}$, Checkout, Pay, Payment Confirmed))

Step 6 – ***Transformation of gates in the threat formula:***

In this step we start transforming the threat formula into a statechart representation by first working on the first gate:

Coupon Fraud = ($\overline{\wedge}$, Signed In, Browse & Select Item, Add Item, Browse or Cart, Add Coupon, Remove Item, Checked out)

Where:

Signed In = ($\vee$, ($\overline{\wedge}$, SignIn, Login, Login Information, Signed), ($\overline{\wedge}$, SignIn, New Customer, Create Account, Signed))

Browse & Select Item = ($\overline{\wedge}$, Browse, Select Item)

Browse or Cart = ($\vee$, Browse, Cart)

Checked out = ($\overline{\wedge}$, Checkout, Pay, Payment Confirmed)

The first step is to represent the threat controller component, which is a Priority AND gate of seven child conditions. Three of them are events and others are gates (sub-goals). Figure 25 shows a statechart representation of the AND gate where inputs that are not events are represented as orthogonal states while inputs that are events are directly represented in the threat controller component. The statechart representation of each simple definition is based on its respective transition in the semantics table. The occurrence of an event will trigger its respective transition in the controller component while the occurrence of the orthogonal states will trigger an action that causes the controller component to change its current state. The OR gate for the *Browse or Cart* condition can be represented as an orthogonal component similar to the representation of the OR gate of or can be directly represented in the controller component. The latter case is used here.

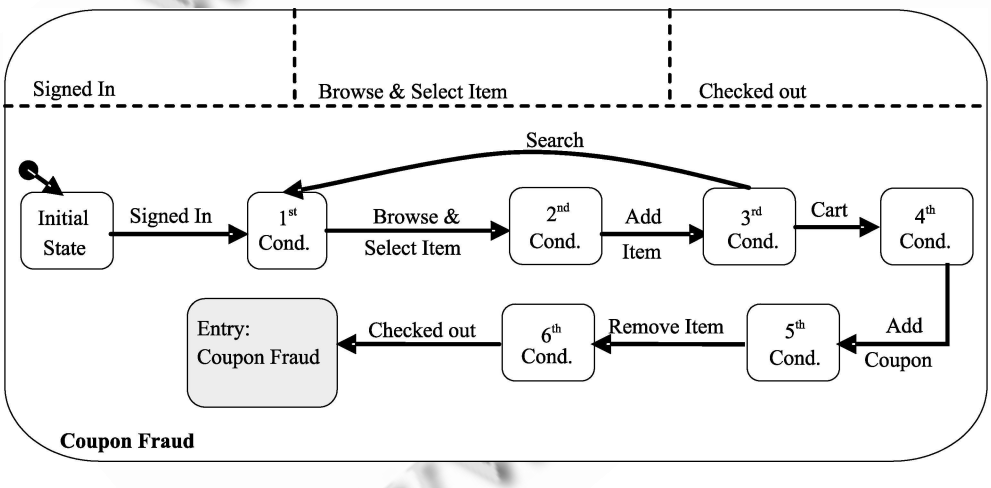

Figure 25. Statechart notation representing the threat controller component

The next step is to have a statechart representation for each orthogonal component. In a security attack there are usually common steps that are shared among other types of attacks. This can be seen in the *Purchase Item at a Reduced Price* attack and the *Coupon Fraud* attack where *Sign In*, *Browse and Select*, and *Checkout*

Copyright by Institute of Software, Chinese Academy of Sciences. All rights reserved

are common in both attacks. Through the use of modular development, where statechart components represent individual steps, shared steps among different attacks can be represented once. An advantage of our approach on attack trees is the capability to reuse components that are common among different attacks. This is exactly the case here where all the three orthogonal components: Signed In, Browse & Select Item, and Checked out have already been given a statechart representation in Fig. 20 respectively.

The last thing to do is to integrate the newly constructed security threat components into the integrated shopping cart behavioral model of Fig. 20. In this case, only the threat controller component is added. Next, the functional behavior of the system should be modified to indicate the occurrence of the attack. This is done by adding a transition to the shopping cart component. This transition will be triggered when the yellow colored state in the threat controller component is reached, and a Coupon Fraud action is emitted. The occurrence of this situation means that the attacker's ultimate goal has been achieved and the threats in the system have been exploited. Figure 26 shows the modified *Shopping Cart* component.

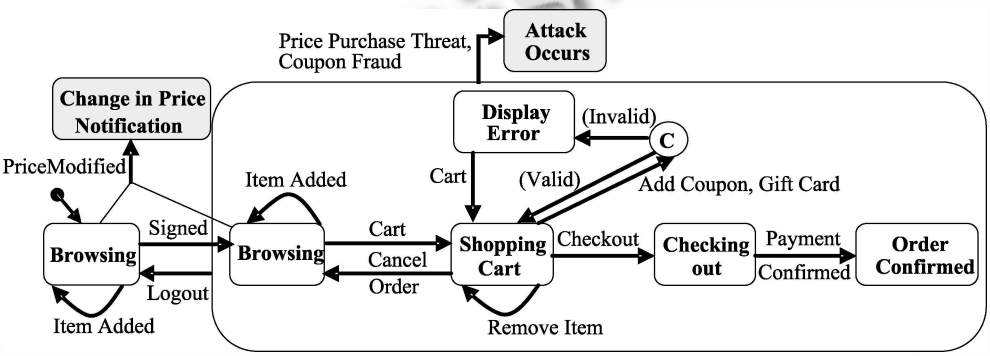

Figure 26. Modified shopping cart component

Step 7 – ***Verification – Identification of Vulnerabilities:***

The integration of a security attack into the system functionalities allows either the verification of threat absence (the attack cannot be achieved) or threat presence (the attack can be completed successfully) in the system. In the case of the Coupon Fraud attack, the integrated components indicate a threat presence. This can be seen from the *threat controller* component of Fig. 25, where all the steps required to complete the attack are allowed through the used system functionalities. The *Shopping Cart* component validates the coupon the moment it is added to the cart. This is seen in Fig. 25 where the moment the 5^{th} *Cond.* state is reached the coupon is not validated anymore. This means that the coupon will not be validated again when the shopping cart is modified (6^{th} *Cond.*) or when the customer completes the checkout process (the yellow state, thus indicating the success of the attack). We can see from the threat behavior that the cause of vulnerability is the lack of coupon validation. This vulnerability is seen in two system components: (1) the *Shopping Cart* component where coupon validation is not done when the cart is updated, (2) the *Checkout* component, where the coupon can be validated again to insure that the coupon conditions are met.

Copyright by Institute of Software, Chinese Academy of Sciences. All rights reserved

Step 8 – ***Mitigation of system vulnerabilities:***

After the vulnerabilities in the system behavior are identified, the next step is to correct these vulnerabilities in order to have a secure model. Coupon validation can either be done on the *Shopping Cart* component or the *Checkout* component to prohibit the fraud of Fig. 25 from occurring:

1. The first approach for mitigation makes sure that when a coupon is added to the cart it will be verified not only the first time it was added but every time the cart gets modified. The focus of this mitigation process will be on the *Remove Item* transition in the *Shopping Cart* component.

2. The second approach delays the process of coupon validation until the customer proceeds with the checkout process. In this case coupons will only be allowed to be added, and thus validated only once, during checkout. This mitigation approach will move the functionalities of adding and verifying a coupon from the *Shopping Cart* component to the *Checkout* component.

The choice of mitigation is usually influenced by the cost of eliminating the system vulnerability. An advantage of our approach is that it allows a better understanding of how the elimination of vulnerability might have effect on multiple threats. It also identifies the components that are needed to be modified during the mitigation process. Mitigations can then be selected to limit the cost and the number of system components to be modified. In case of the attacks presented here, a security architect might choose to add security features to the *Checkout* component in order to eliminate both attacks from occurring in order to decrease the cost of threat mitigation.

Here we will consider the second approach for threat mitigation. In order to mitigate the coupon fraud threat, the functionality of adding and validating coupons should be part of the checkout process. Figure 27 shows the modified Checkout component.

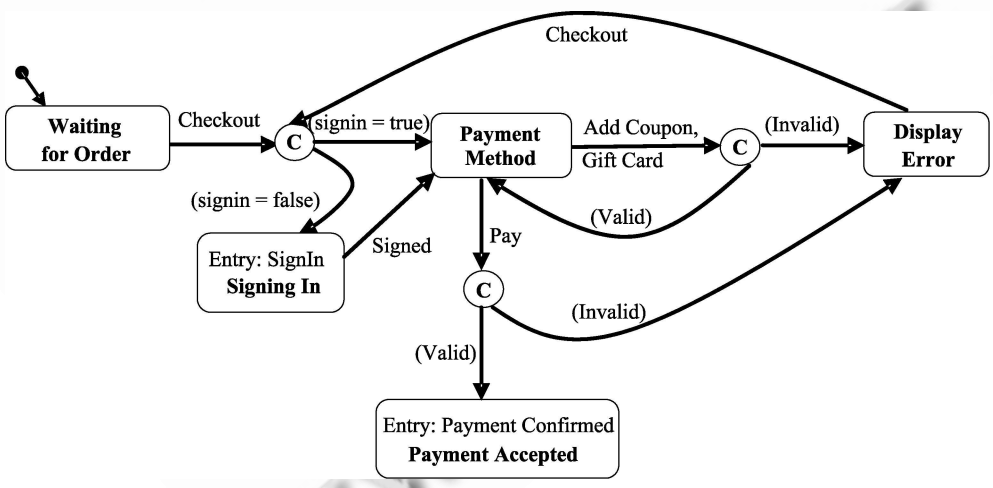

Figure 27. Modified checkout component

The last step, as shown in Fig. 28, is to remove the functionality of adding a coupon from the *Shopping Cart* component.

Copyright by Institute of Software, Chinese Academy of Sciences. All rights reserved

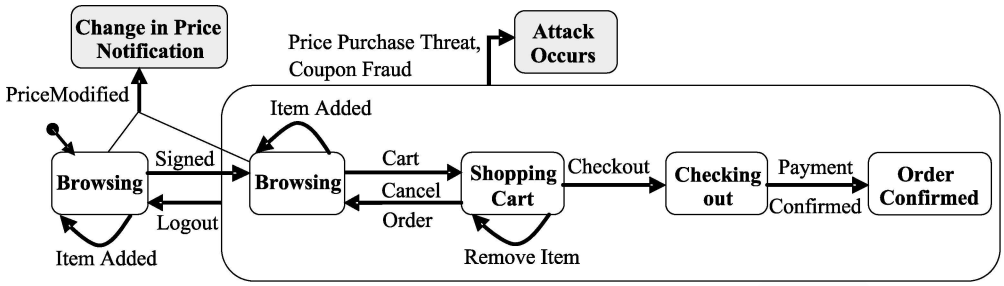

Figure 28. Modified shopping cart component

6.3 Discussion and summary

The case study described the integration of two threats into the shopping cart statechart of Fig. 9. Both of the security attacks in Fig. 10 and Fig. 24 describe the possible ways (required steps) to achieve these types of attacks. However, these attack trees do not focus on the behavior of the system. It is not clear which functional components from Fig. 9 are vulnerable and taken advantage of by the attacker. In addition, it is not clear how the system behaves while the attack is taking place. Finally, it is not clear which actions map to the functionalities of the system and which are external functionalities. That it is why we can understand from Fig. 10 how to buy an item at a reduced price but cannot easily relate this reasoning to the shopping cart statechart or the system functionalities. Similarly, we can understand from Fig. 9 the functional behavior of the shopping cart but cannot understand how these functionalities can be exploited in order to attack the system.

Through the process of threat integration into the shopping cart statechart, software engineers will better understand the system. Their knowledge of the system will not only be based on the correct behavior but also on a deeper knowledge of how the components affect and get affected by security attacks. The process identifies all the system functionalities that are used by the attacker and the threats or vulnerabilities in the software components that were taken advantage of during the attack. Through the integrated model of Fig. 20, the security attack can be easily identified and understood through its interaction with the system functionalities. In addition, threat components that are not part of the functionalities of the system (components that belong to Group 2 or Group 3) are easily identified. It can be seen from Fig. 20 that the *Open HTML Source Code*, *Manipulate HTML Code* and *JavaScript Injection* components belong to group 2 while the *Modify Price* component belongs to Group 3. The rest of the threat components belong to the behavior of the shopping cart system.

Through this integrated model, security concerns and functional behavior are represented together in one model. In addition, the correct behavior of the system is separated from the threat behavior. Only minor modifications were done to the functional behavior. This can be seen in Fig. 20 of the shopping cart case study. Only one action event and a failure state were added to the functional component. Therefore, the functional behavior of the shopping cart is not affected or altered during the integration of the attack. The newly added security attack components can also be easily spotted from the statechart. This improves the understanding of

Copyright by Institute of Software, Chinese Academy of Sciences. All rights reserved

separated interest, such as focusing only on the functional or security threat behavior of the shopping cart system.

Other beneficial observations we found from the case study: (1) Attack trees are not semantically well defined. In other words the description of a security attack through an attack tree can have different interpretations and might be difficult to understand. This lack of a semantically sound foundation can be seen from the changes that underwent the threat formula from the beginning of the integration process (step 1) to the end of the process (the start of step 6). (2) Our approach is capable of reusing components that are common among different attacks. In a security attack, there are common steps that are shared among other types of attacks. Through the use of modular development, where statechart components represent individual steps, shared steps among different attacks can be represented once. (3)The choice of mitigation is usually influenced by the cost of eliminating the system vulnerability. An advantage of our approach is that it allows a better understanding of how the elimination of vulnerability might have effect on multiple threats. It also identifies the components that are needed to be modified during the mitigation process. Mitigations can then be selected to limit the cost and number of components that are needed to be modified.

As for the scalability of our proposed approach, it largely depends on the scalability of attack trees and statecharts. Our proposed approach provides a deeper look into an attack and shows how the behavior of a system is affected by this attack. Furthermore, the integrated model represents the low level behavior of a threat, and as a result the approach is semantically and descriptively richer than attack trees. Naturally, dealing with a richer, lower level description of a failure implies additional complexity to its representation, such as exponential explosion of the number of states in the integrated model. Our choice for using statecharts is due to its capability to avoid this state explosion problem through the use of hierarchical decomposition and concurrency. Another concern is the applicability of our approach to attack trees with large number of nodes. A complex tree can be handled by first dividing the tree into independent modules, and then working on each module separately as if they were different attack trees.

7 Conclusions

We have presented an approach that deals with security attacks in the system modeling process. It focuses on representing each security threat precisely from the system's perspective. The integration process tries to eliminate the semantic ambiguities between the statecharts and attack trees. The result is a system model that includes the intended functional behavior, the behavior of achieving an attack successfully, and the behavior of the system during an attack. From this model, vulnerabilities are then identified and mitigated. The case study has demonstrated the different aspects of attack tree notations and how they can be integrated into system statecharts.

The main motivation for this work was to improve, strengthen and precisely define security concerns and their effect on software during the design phase. This will allow security engineers to collaborate with software engineers on the software design. This collaboration is crucial for software engineers to clearly understand

Copyright by Institute of Software, Chinese Academy of Sciences. All rights reserved

security attacks and to determine threats and vulnerabilities in the system. As it is the case that our proposed work integrates two models that are heterogeneous in both structure and semantics, there might still be a need for human interventions to clarify the mismatches and ambiguities between the two models. The presence of a security architect or specialist might still be required during the integration process. A natural direction for extending this work is to develop a tool for managing and assisting the collaboration between the security and system engineers. Other directions of future work include deriving security metrics from the integrated model and generating security attacks for penetration testing.

References

[1] Wysopal C,Nelson L, Dai Zovi D, Dustin E. The art of software security testing: identifying software security flaws. Addison-Wesley Professional, 2006.

[2] Xu D. Software security. Wiley Encyclopedia of Computer Science and Engineering. John Wiley & Sons, 2009.

[3] Torr P. Demystifying the threat-modeling process. IEEE Security and Privac, Sep. 2005, 3(5): 66–70.

[4] Howard M, LeBlanc D. Writing Secure Code. Microsoft Press, 2003.

[5] Swiderski F, Snyder W. Threat Modeling. Microsoft Press, 2004.

[6] Schneier B. Attack Trees. Dr. Dobb's Journal of Software Tools, Dec. 1999, 24(12): 21–29.

[7] El Ariss O, Xu D. Modeling security attacks with statecharts. Proc. of the 2nd International ACM Sigsoft Symposium on Architecting Critical Systems (ISARCS 2011). Boulder, USA. June 2011. 20–24.

[8] El Ariss O, Xu D, Wu J. Building a secure system model: integrating attack trees with statecharts. Proc. of the 5th International Conference on Secure Software Integration and Reliability Improvement (SSIRI 2011). Korea. June 2011. 27–29.

[9] Viega J, McGraw G. Building Secure Software: How to Avoid Security Problems in the Right Way. Addison Wesley, 2002.

[10] Landwehr CE. Formal models for computer security. Computing Surveys, Sep. 1981, 13(3): 247–277.

[11] Allen J, et al.. State of the Practice of Intrusion Detection Technologies. Technical Report CMU/SEI-99-TR-028, 2000.

[12] Weissman C. Penetration Testing. In: Abrams M.D., Jajodia, S., Podell, H. eds. Information Security Essays. IEEE Computer Society Press, 1994.

[13] Helmer G, Wong J, Slagell M, Honavar V, Miller L, Lutz R. A software fault tree approach to requirements analysis of an intrusion detection system. Requirements Eng., 2002, 7(4): 177–220.

[14] Alexander I. Misuse cases: use cases with hostile intent. IEEE Software. 2003: 58–66.

[15] Xu D, Pauli J. Threat-driven design and analysis of secure software architectures. Journal of Information Assurance and Security, 2006, 1(3): 171–180.

[16] Xu D, Goel V, Nygard K, Wong WE. Aspect-oriented specification of threat-driven security requirements. International Journal of Computer Applications in Technology, Special Issue on Concern Oriented Software Evolution, 2008, 31: 131–140.

[17] Van Lamsweerde A. Elaborating security requirements by construction of intentional anti-models. Proc. Int'l Conf. Software Eng., 2004. 148–157.

[18] McDermott J. Attack Net Penetration Testing. The 2000 New Security Paradigms Workshop. ACM SIGSAC, Sept. 2000: 15–22.

[19] Xu D, Nygard KE. Threat-driven modeling and verification of secure software using aspect-oriented petri nets. IEEE Trans. on Software Engineering, 2006, 32(4): 265–278.

[20] Schumacher M, Haul C, Hurler M, Buchmann A. Data mining in vulnerability databases. 7th Workshop Sicherheit in Vernetzten Systemen. Germay, March 2000.

[21] Common Criteria for Information Technology Security Evaluation, Version 3.1 Revision 3, July 2009 http://www.commoncriteriaportal.org/thecc.html.

Copyright by Institute of Software, Chinese Academy of Sciences. All rights reserved

[22] Schumacher M. Security Engineering with Patterns: Origins, Theoretical Models, and New Applications. Springer-Verlag 2003.
[23] Ray I, Li N, Kim D, France R. Using parameterized UML to specify and compose access control models. Proc. of the 6th Conf. on Integrity and Internal Control in Information Systems. 2003.
[24] Mouheb D, Talhi C, Lima V, Debbabi M, Wang L, Pourzandi M. Weaving security aspects into UML 2.0 design models. Proc. of the 13th Workshop on Aspect-Oriented Modeling. 2009.
[25] Pavlich-Mariscal J, Michel L, Demurjian S. Enhancing UML to Model Custom Security Aspects. Proc. of AOM@AOSD, 2007.
[26] Jurjens J. Using UMLsec and goal trees for secure systems development. Proc. of SAC'02. 2002.
[27] Kong J, Xu D. A UML-based framework for design and analysis of dependable software. Proc. of COMPSAC'08. 2008.
[28] El-Ariss O, Xu D,Wong WE, Chen Y, Lee Y. A systematic approach for integrating fault trees into system statecharts. Proc. of COMPSAC'08. July 2008.
[29] U.S. Nuclear Regulatory Commission, Fault Tree Handbook. NUREG-0492. Washington, D.C., January 1981.
[30] Mauw S, Oostdijk M. Foundations of attack trees. Information Security and Cryptology - Springer Lecture Notes in Computer Science. 2005, 3935: 186–198.
[31] Harel D. Statecharts: A visual formalism for complex systems. Science of Computer Programming, 1987, 8: 231–274.
[32] Drusinsky D. Modeling and Verification Using UML Statecharts. Elsevier, 2006.
[33] Harel D, Politi M. Modeling Reactive Systems with Statecharts: The STATEMATE Approach. McGraw-Hill, 1998.
[34] Von Borstel FD, Gordillo JL. Model-based development of virtual laboratories for robotics over the internet. IEEE Trans. Syst., Man, Cybern. A, Syst., Humans, May 2010, 40(3): 623–634.
[35] McClure S, Shah S, Shah S. Web Hacking: Attacks and Defense. Addison-Wesley, 2003.
[36] Meier JD, Mackman A, Vasireddy S, Dunner M, Escamilla R, Murukan A. Improving Web Application Security-Threats and Countermeasures. Microsoft Corporation, 2003.

www.ijsi.org

Copyright by Institute of Software, Chinese Academy of Sciences. All rights reserved