APPLICATION SPECIFIC USAGE CONTROL IMPLEMENTATION VERIFICATION

Rajkumar P.V.[1], S.K.Ghosh[2] and Pallab Dasgupta[3]

[1]School of IT, Indian Institute of Technology-Kharagpur, India
vrj@sit.iitkgp.ernet.in
[2] School of IT, Indian Institute of Technology-Kharagpur, India
skg@iitkgp.ac.in
[3]Department of CSE, Indian Institute of Technology-Kharagpur, India
pallab@cse.iitkgp.ernet.in

ABSTRACT

Usage control is a comprehensive access control model developed to cater the security needs of the wide range of applications. Formal specification of the core usage control models and their expressivity, decidability of safety properties are explored recently. They help us to understand the usability and safety of the model. However, security of the usage control in the practical applications depends on the safety of the model as well as its correct implementation in the application. This paper presents an approach to verify the correctness of the usage control implementation using a semi- formal property verification tool. We also provide an illustrative case study.

KEYWORDS

Usage Control, Software Implementation, Integrated Verification.

1. INTRODUCTION

Usage control [1] is the most comprehensive access control model and it supports wide range of polices varying from role based on access control to digital rights management. Access decision in the usage control is based the attributes of subjects, objects, system and the environment. These attribute values can be dynamically modified by the independent update actions. The occurrence of one action may influence the permission for the other (as well as the same) actions in future.

The logical specifications [2] of the usage control model and its safety analysis [3] have been explored recently. The safety decidable usage control model can ensure the model level correctness, however correctness of usage control in the software application depends on both the safety of the model as well as the correctness of its implementation in the application. This paper addresses the correctness verification problem of safety decidable usage control [3] implementation. Even though this paper is focused on usage control implementation the proposed approach is commonly applicable for any safety decidable access control model.

An application's usage control implementation consists of primitive functions to manipulate the protection state and the higher level functions which mediate the user interactions in accordance with the specific access control policy. While implementing the usage control model in the applications, the programmers may commit mistakes. The implementation errors may occur in any of the primitive usage control functions which defines and manages the protection state of the application as well as their integration. Note that even if the model is theoretically safe, the errors in the implementation will defy its very purpose of protection. Most of the work in access control research left the correctness of implementation as a general software verification problem. Even though access control implementation is a software module, there is subtle difference between the general software verification and verification of access control implementation. Software verification in general covers all the functional correctness of the system which involves the complete state space of the application, whereas access control implementation of an application revolves over few repeated usage control/protection states. Manually ensuring their correctness is a hard task and the conventional software testing methods are not tuned for exploring the usage control states. This paper proposes an integrated verification approach which explores only the usage control states of the application. The main contributions of the paper can be summarized as follows

1. We present a method to abstract the usage control irrelevant code details and isolate the usage control state space of the application.

2. We illustrate a way to leverage recent developments in the formal property verification to verify the security of application specific usage control implementation.

Remaining part of the paper is organized as follows. Section II provides the related work. Section III presents an application specific usage control implementation. Section IV presents the integrated verification approach for usage control implementation verification. Section V provides an illustrative case study. Section VI presents the conclusion.

2. RELATED WORK

Initial studies on access control and protection had begun in the context of operating systems, where the role of access control was to protect various system resources from unauthorized usage by the processes in the system [4] [5]. In the year 1976, Harrison, Ruzzo, and Ullman [6] developed the abstract model for studying the theoretical soundness of the access control in the computing systems, later it was known as HRU model. Their model consists of high-level abstract representation of protection states and the commands for manipulating those abstract states. The abstract representation models the system resources as objects, the processes which can utilize the resources as subjects, and the different ways in which they can be utilized as access rights. This model of protection is completely free from the implementation details of the various processes running in the system as well as the peculiarities of individual resources. This representation precisely isolates the design of access control from the implementation and facilitates the formal reasoning about the design.

The access control (model) design is said to be safe if it does not allow any unauthorized subject to access the object in the model. Formally proving the safety of access matrix model is shown to be Turing complete [6]. After this negative result, most of the research has been focused on designing access control models with decidable safety property. In the last three decades, many access control models with decidable safety property are proposed [7] [8] [9], [10] [11] [12] [3]. Apart from them, it is also possible to develop application specific access control models with decidable safety. For ensuring software application's access control security, theoretical proof of safety of the underlying model is essential; at the same time, correctness of the implementation is also an equally important factor.

Naldurg et al. [13] present a tool to find the information-flow vulnerabilities by using the access control meta-data i.e. access control entries in the access control list implementation. This tool takes an instance of access control configuration and derives the relational expression through which the information-flow vulnerabilities were identified. In Role Based Access Control Model, user-role and role- permission assignments determine the access privileges of individual users, Hansen et al. [14] and Masood et al. [15] present methods to verify conformance between the access control policy and its respective role-permission, the user-role assignments. Sen et al. [16] and Brucker et al. [17] describe methods to test the correctness of access control implementation in fire-walls. Guelev et al. [18] presents model checking approach for verifying the consistency of access control policy specification and Zhang et al. [19] presents a method to synthesize XACML tags from the verified policy. Martin et al. [20] presents a method to test XACML policies using fault model. Traon et al. [21] used mutation analysis for access control test case generation. Pretschner et al. [22] presents the Z-specification of mechanisms for usage control models. Mallouli et al. [23] presents a way to integrate security policy with the system design specification. Hu et al. [24] [25] presents methods to verify the correctness of access control constraint specification and implementation. This paper presents an application specific safety decidable usage control implementation verification.

3. Application Specific Usage Control Implementation

Software applications have a large number of states due to complex data variables and the unrestricted functional manipulations of these states produce a complex state transition relation among them. Often, they are not amenable for formal analysis. However, the section of the code meant for usage control of an application involves relatively simple date variables—most often they are Boolean valued—which constitute the protection state space of the application.

Definition 1 Protection State Variables are defined as *the set of variables whose values define the usage rights of the users in the application.*

Definition 2 Protection State Space is defined as *all possible valuations of protection state variables in the application.*

Protection state variables and their valuations along with the usage control policy determine the usage rights of the user. These variables tend to change over time to provide the dynamism in the usage control of the application. However, the functions which are allowed to operate on these variables are bounded by the usage control model tailored for the application. These functions determine the range of dynamism supported by the usage control system.

Definition 3 Primitive usage Control Functions are defined as *the set of functions which are directly manipulating the protection state variables and make the state transitions within the protection state space.*

The protection states are defined and maintained by primitive usage control functions and the remaining functional part of the implementations are not directly involved with the protection state; instead, they use protection states to control their execution in accordance with the usage control policy. Hence the usage control implementation verification can safely abstract functional details except the information about integration of primitive usage control functions with the functional parts.

3.1. Action Based Abstraction

Software applications consist of different interfaces and the methods called from those interfaces. Here usage control bits are used for synchronization between the interfaces by implementing application specific usage constraints. For example, in a typical conference management application, submission of paper by an author automatically disqualifies him from acting as a reviewer of the same paper in future. Note that submission interface and reviewer interface are typically separate functional modules that synchronize through the role-based usage control policy enforced by the application.

The fundamental model of such computation can be represented as set of actions and each action is guarded with pre and post conditions.

The guarding pre-conditions synchronize various usage control sensitive actions in the application based on the valuation of usage control bits; the update bits capture the changes in usage control configuration due to the action.

Syntax of actions: < Pre-conditions > Action code < Updation of usage control bits >

Usage controls of an application co-ordinate only the functional modules, which use the resources protected by usage control policy. Remaining parts of the application are not sensitive to the usage control which includes the authentication and other security modules. For example, user registration module of the conference management application authenticates subjects in the usage control but the function of the module is out of the usage control's protection.

Definition 4 Usage Control Sensitive Actions are defined as *the functions whose execution changes the usage control configuration or those functions whose execution is guarded with usage control permissions.*

The usage control sensitive actions are model independent; they are defined with respect to the specific application and its usage control policy. Finding the precise set of usage control sensitive actions in the application is also an important factor to ensure the correctness of usage control implementation in the final product. The imprecise set may lead to uncontrolled usage of usage control policy protected resources. Usage control sensitive actions can be broadly classified as follows

- **Request Actions** are those actions that need usage control mediation in the application. Usage control configuration determines whether the request can be granted, also it notifies the result of the request by invoking the corresponding response action. These actions do not make any changes in usage control configuration.

- **Update Actions** are generated within the application environment, like time based triggers or special action triggered by cumulative effect of user actions. These actions change usage control configurations but to perform the action they may not to have explicit permission.

- **Request-Update** Actions are almost same as *Request* actions; the only difference is that the successful completion of the *Request-Update* action makes changes in usage control configurations.

Usage control configuration changes in accordance with the sequence of usage control sensitive actions occurred over time. All possible configurations of the usage control forms the protection state space of the application. The usage control implementation verification is focused on verifying the correctness of the source code which manipulates the protection state space.

4. An Integrated Approach for Usage Control Implementation Verification

4.1 Protection State Modelling

Given the action based abstraction, all possible changes in the protection states can be modelled as a state transition system in which the protection states are the states and the usage control sensitive actions along with the *precondition* and *updates* are the transition labels. It can be formally defined as follows

Definition 4 Protection State Transition System PSTS *is defined as a six-tuple PSTS = (Q, ∑, T, C, U, I) where,*

- *Q is a finite set of protection states,*
- *∑ is a finite set of access control sensitive actions,*
- *C is a finite set of conditions,*
- *U is a finite set of update actions,*
- $T \subseteq Q \times \sum \times C \times 2^U \times Q$ *is a finite set of transitions,*
- *I Є Q is the initial state.*

Usage control is a reactive system so the protection state transition system does not have any final state. This model isolates the protection state space of the application from the rest. This isolation and the nature of the protection state space makes the usage control implementation verification different from general software verification.

For example, consider the conference management system application with a usage control policy like after submitting a paper, the author is not allowed to view his paper's review report until the notification of acceptance. At the implementation of this application, *SubmitPaper* action sets the *ReadReviewReport-right* to false and the *AcceptanceNotified* action may set the *ReadReviewReport-right* to true. The applications state space consist of valuation of all the variables present in the source code of *SubmitPaper*, *Read-ReviewReport, AcceptanceNotified* functions and their GUI front-ends. However, *ReadReviewReport-right* is the only Boolean variable which forms the protection state space. And, the protection state transition system consist of two states namely false state and true state, there is a self-loop with *SubmitPaper* label in false state and there is transition from *false* state to *true* state with the *AcceptanceNotified* label. This model is significantly less complex compared to the remaining state space of the application. Protection state model captures all possible valuation of usage bits and abstracts the rest of the code details. The verification method, given in the figure-1, explores only this isolated protection state space for checking the correctness of usage control implementation.

4.2 Property Specification

The verification needs the formal specification of the correctness properties. Property specification languages widely used in the formal verification community are predominantly either state-based or action-based. To describe correctness properties for usage control implementation, one needs to describe properties over actions and their parameters as well. Hence the proposed method uses Action LTL language [26] for policy specification. Action LTL extends Linear Temporal Logic (LTL) with some interesting features like the ability to express properties over actions, ability to express arithmetic and relational queries over data attributes (both Boolean and integer valued), the concept of local variables and the concept of parameterized actions. The syntax [26] of action-LTL is given as follows

Let 'p' be a predicate over data variables and 'e' be an event dispatched to the subject

- *p* is an action-LTL formula.
- *e* is an action-LTL formula.

- If f and g are action-LTL formulas, then so are ~f, f and g, f or g, Xf, Gf, Ff, f U g.

The following sections present the practical illustration of protection-state modelling and verification using the well-known conference management system application. For verification purpose, we use the integrated verification framework presented in [27]. The constrained random test case generator may use the methods similar to the methods described in [28], [29] to generate usage control specific test cases.

4.3 Verification

The verification method consists of the protection state model of the application, constrained test generator and a dynamic verification tool as depicted in the figure 1. The m1, m2 and m3 are concurrent protection state transition systems of the application. The constrained test generator is used to generate and execute usage control specific test cases. The dynamic verification tool observes the events occurred in the application and checks for the property violations.

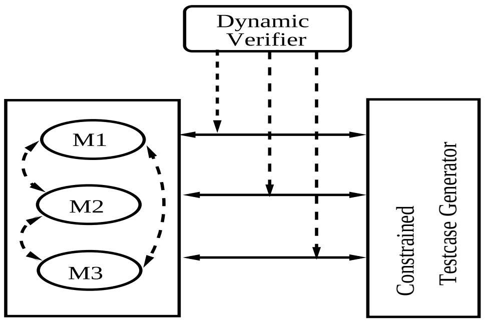

Figure 1. Formal property verification method

This method is easily adoptable to the industry level UML based software development tool suits like the IBM Rational Rhapsody [30]. The typical medium sized application developed in high-level languages, like Java and C++, have thousands of lines source code. It is hard to define and track the usage control relevant sampling points, which forms the protection state space, at the source code level. Instead, the UML state-machines facilitate the modelling of protection state transition system at higher level of abstraction i.e. it allows us to define the states and the actions [31]. The implementation code can be embedded within the states. Naturally the states provide the hooks to various interface points in the source code. The actions, which cause the protection state transitions can also be captured and guarded with pre- and post-conditions.

The industry standard UML tool suites like IBM Rational Rhapsody [30] are shipped with action based simulation environments, which provides a clear route to examine the state of the system after the occurrence of each action. Hence, the proposed method makes use of the UML state machines to model the protection state transition systems.

5. CASE STUDY

We illustrate the above mentioned approach with the case study, similar to the one used in [18]. Consider a typical conference management application with the following usage control policy requirements

1. Authors can submit the papers and they should be excluded from reviewing their own paper.

2. After the submission, the program committee members can randomly choose papers to review.

3. Reviewers are allowed to write and submit the reports without exceeding 250 words. After submission, no modification should be allowed.

4. Reviewers can read other's report after submitting their own report and after reading the report, they should not be allowed to submit a report about the same paper.

5. After reading the reviewer's report, the chair is allowed to make the acceptance decision of a paper.

6. After making the acceptance decision, authors can view the acceptance status of their paper and copy their review reports.

5.1 Safe Usage Control Model Design

We use the safety decidable fragments [3] of usage control [2] to model the usage control policy needs. It has the following basic elements

- **Subjects:** Authors, Program Committee (PC) Members and the Chair.

- **Objects:** Papers and Review Reports.

- **Rights:** submit, read, write, take-reviewership, make-decision and copy.

The subjects and the objects have set of finite valued attributes. Authorization predicates and the update functions are defined using these attribute values. Usage of rights, except the copy right, makes changes in usage rights of the model. For example, usage of *take-reviewership* right enables the user to write and submit review report. These changes are captured in pre-update functions of the model. The usage control model tailored for this application needs is given in figure-2.

The initial configuration of the model has finite number of subjects and each subject has unique identifier. The fixed number of subjects has *PCMembership* and one subject has the *ChairMembership*. The usage control policies *permitaccess(s,o,submit)* and *permitaccess(s,o, takereviewership)* creates objects in the model. This specification remains within the safety decidable fragments of the usage control authorization model.

5.1 Protection State Modelling

We used the UML based application development suite IBM Rational Rhapsody to implement the application. The subjects and objects in the usage control model are implemented as instance of classes and the usage rights are coded as symbolic constants. The actions in the usage control model like *tryaccess*, *predicate functions*, and *update functions* are implemented as overloaded member functions of usage control class. These functions are called the primitive usage control functions. Rest of the application uses these primitive usage control functions to implement the functional level usage control requirements. Note that every usage control sensitive user action invokes a primitive usage control function at least once in its execution cycle.

$$\begin{array}{rcl}
\mathit{permitaccess(s,o,submit)} & \rightarrow & \blacklozenge\mathit{(tryaccess(s,o,submit)} \wedge \mathit{IsPaperType}_a\mathit{(o))} \wedge \blacklozenge\mathit{preUpdate(o.ID)} \\
 & & \wedge\blacklozenge\mathit{preUpdate(o.Author)} \wedge \blacklozenge\mathit{preUpdate(s.PaperIDList)} \\
\mathit{IsPaperType}_a\mathit{(o)} & : & \mathit{o.Type} = \mathit{Paper} \\
\mathit{preUpdate(o.ID)} & : & \mathit{o.ID} \leftarrow \mathit{CreateNewObject(Type{:}Paper)} \\
\mathit{preUpdate(o.Author)} & : & \mathit{o.Author} \leftarrow \mathit{s.ID} \\
\mathit{preUpdate(s.PaperIDList)} & : & \mathit{s.PaperIDList'} \leftarrow \mathit{AppendList(s.PaperIDList,o.ID)}
\end{array}$$

$$\begin{array}{rcl}
\mathit{permitaccess(s,o,takereviewership)} & \rightarrow & \blacklozenge\mathit{(tryaccess(s,o,takereviewership)} \wedge \mathit{ReviewReportNotRead}_a\mathit{(s,o)} \\
 & & \wedge\mathit{IsPaperType}_a\mathit{(o)} \wedge \mathit{IsPCMember}_a\mathit{(s)} \wedge \mathit{IsNotAuthor}_a\mathit{(s,o))} \\
 & & \wedge\blacklozenge\mathit{preUpdate(o.ReviewersList)} \ \wedge \blacklozenge\mathit{preUpdate(s.ReviewPapersList)} \\
 & & \wedge\blacklozenge\mathit{preUpdate(o.ReportID)} \wedge \blacklozenge\mathit{preUpdate(s.ListOfReportsRead)} \\
\mathit{ReviewReportNotRead}_a\mathit{(s,o)} & : & \mathit{o.ID} \notin \mathit{s.ListOfReportsRead} \\
\mathit{IsPCMember}_a\mathit{(s)} & : & \mathit{PCMembership} \in \mathit{s.Memberships} \\
\mathit{IsNotAuthor}_a\mathit{(s,o))} & : & \mathit{s.ID} \notin \mathit{o.Authors} \\
\mathit{preUpdate(s.ReviewPapersList)} & : & \mathit{s.ReviewPapersList'} \leftarrow \mathit{AppendList(s.ReviewPapersList,o.ID)} \\
\mathit{preUpdate(o.ReviewersList)} & : & \mathit{o.ReviewersList'} \leftarrow \mathit{AppendList(o.ReviewersList,s.ID)} \\
\mathit{preUpdate(o.ReportID)} & : & \mathit{o.ReportID} \leftarrow \mathit{CreateNewObject(Type{:}Report,Paper{:}o.ID,Reviewer{:}s.ID)} \\
\mathit{preUpdate(s.ListOfReportsRead)} & : & \mathit{s.ListOfReportsRead'} \leftarrow \mathit{AppendList(s.ListOfReportsRead,o.ID)}
\end{array}$$

$$\begin{array}{rcl}
\mathit{permitaccess(s,o,submit)} & \rightarrow & \blacklozenge\mathit{(tryaccess(s,o,submit)} \wedge \mathit{IsReportType}_a\mathit{(o)} \wedge \mathit{IsReviewer}_a\mathit{(s,o)} \\
 & & \wedge\neg\mathit{IsReportSubmitted}_a\mathit{(o)} \wedge \mathit{IsReportWithInLimit(o))} \wedge \\
 & & \blacklozenge\mathit{preUpdate(o.ReportSubmitted)} \wedge \blacklozenge\mathit{preUpdate(s.ReportSubmitted)} \\
\mathit{IsReportType}_a\mathit{(o)} & : & \mathit{o.Type} = \mathit{Report} \\
\mathit{IsReviewer}_a\mathit{(s,o)} & : & \mathit{s.ID} = \mathit{o.Reviewer} \\
\mathit{IsReportSubmitted}_a\mathit{(o)} & : & \mathit{o.Submitted} = \mathit{True} \\
\mathit{IsReportWithInLimit}_a\mathit{(o)} & : & \mathit{o.WordCount} \leqslant 250 \\
\mathit{preUpdate(o.ReportSubmitted)} & : & \mathit{o.ReportSubmitted} \leftarrow \mathit{True} \\
\mathit{preUpdate(s.ReportSubmitted)} & : & \mathit{s.ReportSubmitted} \leftarrow \mathit{True}
\end{array}$$

$$\begin{array}{rcl}
\mathit{permitaccess(s,o,write)} & \rightarrow & \blacklozenge\mathit{(tryaccess(s,o,write)} \wedge \mathit{IsReportType}_a\mathit{(o)} \wedge \mathit{IsReviewer}_a\mathit{(s,o)} \\
 & & \wedge\neg\mathit{IsReportSubmitted}_a\mathit{(o)} \wedge \mathit{IsReportWithInLimit}_a\mathit{(o))} \\
\mathit{permitaccess(s,o,write)} & \rightarrow & \Diamond\mathit{(onUpdate(o.WordCount)} \wedge \Diamond\mathit{endaccess(s,o,write))} \\
\mathit{onUpdate(o.WordCount)} & : & \mathit{o.WordCount} \leftarrow \mathit{TokenCount(o.Report,\text{" "})}
\end{array}$$

$$\begin{array}{rcl}
\mathit{permitaccess(s,o,read)} & \rightarrow & \blacklozenge\mathit{(tryaccess(s,o,read)} \wedge \mathit{IsPaperType}_a\mathit{(o)} \wedge \mathit{IsReviewer}_a\mathit{(s,o))} \\
\mathit{permitaccess(s,o,read)} & \rightarrow & \blacklozenge\mathit{(tryaccess(s,o,read)} \wedge \mathit{IsReportType}_a\mathit{(o)} \wedge \mathit{((IsPCMember}_a\mathit{(s)} \wedge \\
 & & \mathit{IsReportSubmitted}_a\mathit{(s))} \vee \mathit{IsChair}_a\mathit{(s))} \wedge \mathit{IsReportSubmitted}_a\mathit{(o)} \\
 & & \wedge\mathit{IsNotAuthor}_a\mathit{(s,o))} \wedge \blacklozenge\mathit{preUpdate(s.ListOfReportsRead)} \\
\mathit{IsChair}_a\mathit{(s)} & : & \mathit{s.ID} = \mathit{Chair}
\end{array}$$

$$\begin{array}{rcl}
\mathit{permitaccess(s,o,make\text{-}decision)} & \rightarrow & \blacklozenge\mathit{(tryaccess(s,o,make\text{-}decision)} \wedge \mathit{IsChair}_a\mathit{(s)} \wedge \mathit{ReviewReportRead}_a\mathit{(s,o))} \\
 & & \wedge\blacklozenge\mathit{preUpdate(o.DecisionMade)} \\
\mathit{preUpdate(o.DecisionMade)} & : & \mathit{o.DecisionMade} \leftarrow \mathit{True} \\
\mathit{permitaccess(s,o,copy)} & \rightarrow & \blacklozenge\mathit{(tryaccess(s,o,copy)} \wedge \mathit{IsAuthor}_a\mathit{(s,o)} \wedge \mathit{IsDecisionMade}_a\mathit{(o))}
\end{array}$$

Figure 2 Conference management application's usage control model specification

The primitive usage control functions co-ordinate the usage control sensitive actions in accordance with usage control policy. Usage control policy defines the acceptable protection states and their legal transitions.

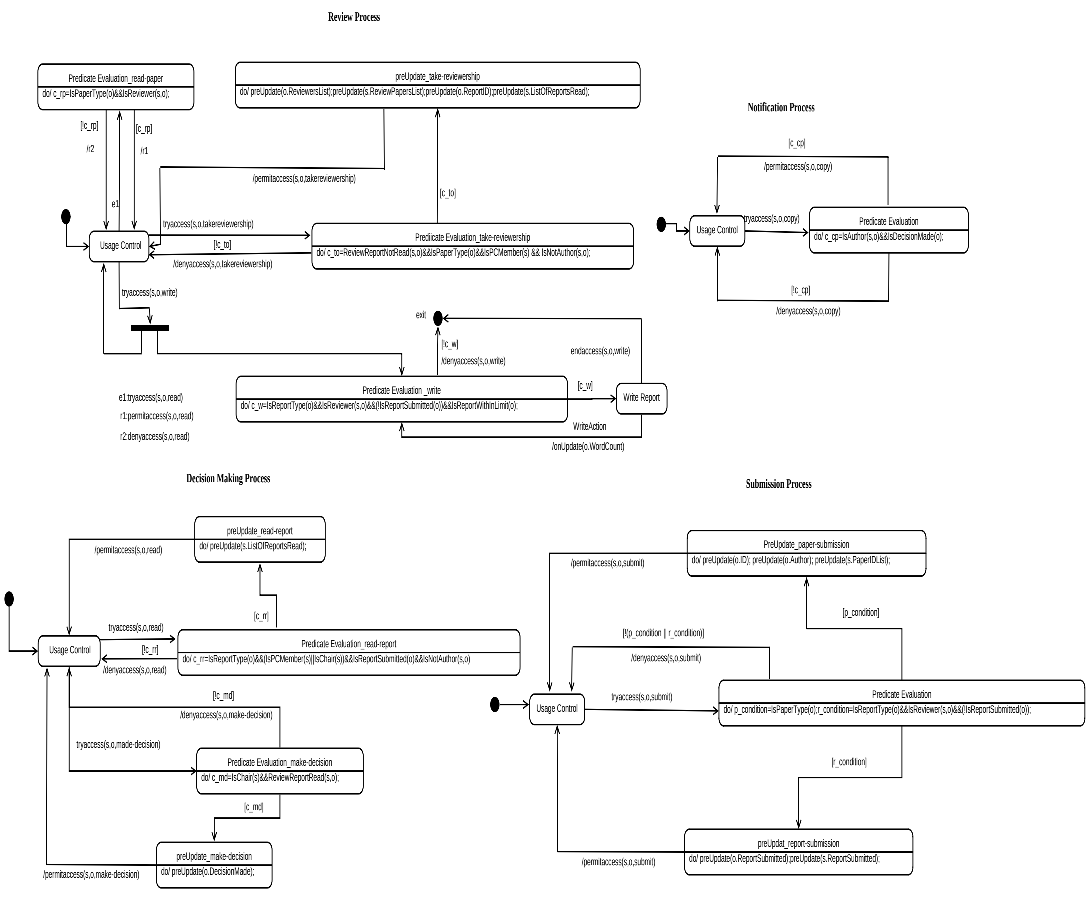
Review Process
Predicate Evaluation_read-paper
do/ c_rp=IsPaperType(o)&&IsReviewer(s,o);
preUpdate_take-reviewership
do/ preUpdate(o.ReviewersList);preUpdate(s.ReviewPapersList);preUpdate(o.ReportID);preUpdate(s.ListOfReportsRead);
[!c_rp]
/r2
[c_rp]
/r1
/permitaccess(s,o,takereviewership)
[c_to]
e1
tryaccess(s,o,takereviewership)
Usage Control
[!c_to]
/denyaccess(s,o,takereviewership)
Prediicate Evaluation_take-reviewership
do/ c_to=ReviewReportNotRead(s,o)&&IsPaperType(o)&&IsPCMember(s) && IsNotAuthor(s,o);
tryaccess(s,o,write)
exit
[!c_w]
/denyaccess(s,o,write)
endaccess(s,o,write)
e1:tryaccess(s,o,read)
r1:permitaccess(s,o,read)
r2:denyaccess(s,o,read)
Predicate Evaluation _write
do/ c_w=IsReportType(o)&&IsReviewer(s,o)&&(!IsReportSubmitted(o))&&IsReportWithInLimit(o);
[c_w]
Write Report
WriteAction
/onUpdate(o.WordCount)
Notification Process
[c_cp]
/permitaccess(s,o,copy)
Usage Control
tryaccess(s,o,copy)
Predicate Evaluation
do/ c_cp=IsAuthor(s,o)&&IsDecisionMade(o);
[!c_cp]
/denyaccess(s,o,copy)
Decision Making Process
preUpdate_read-report
do/ preUpdate(s.ListOfReportsRead);
/permitaccess(s,o,read)
[c_rr]
tryaccess(s,o,read)
Usage Control
[!c_rr]
/denyaccess(s,o,read)
Predicate Evaluation_read-report
do/ c_rr=IsReportType(o)&&(IsPCMember(s)||IsChair(s))&&IsReportSubmitted(o)&&IsNotAuthor(s,o)
[!c_md]
/denyaccess(s,o,make-decision)
tryaccess(s,o,made-decision)
Predicate Evaluation_make-decision
do/ c_md=IsChair(s)&&ReviewReportRead(s,o);
[c_md]
preUpdate_make-decision
do/ preUpdate(o.DecisionMade);
/permitaccess(s,o,make-decision)
Submission Process
PreUpdate_paper-submission
do/ preUpdate(o.ID); preUpdate(o.Author); preUpdate(s.PaperIDList);
/permitaccess(s,o,submit)
[p_condition]
[!(p_condition || r_condition)]
/denyaccess(s,o,submit)
Usage Control
tryaccess(s,o,submit)
Predicate Evaluation
do/ p_condition=IsPaperType(o);r_condition=IsReportType(o)&&IsReviewer(s,o)&&(!IsReportSubmitted(o));
[r_condition]
preUpdat_report-submission
do/ preUpdate(o.ReportSubmitted);preUpdate(s.ReportSubmitted);
/permitaccess(s,o,submit)

Figure 3 Usage control implementation model of conference management application

Usage control of the application is modelled as four concurrent reactive threads namely submission process, review process, decision making process and notification process. Among them review process involves the ongoing control for *write* right; hence we implemented a concurrent sub-thread to process each *tryaccess* action. The state machine model of the conference management application's usage control is given in figure 3.

All the state machines start with usage control state and depending on the *type of tryaccess* action; it makes the transition to the predicate evaluation state. If the guarding predicates are satisfied then it moves the *preUpdate* state and returns to the usage control state, while returning it generates the respective *permitaccess* action; else, it returns to usage control state, while returning it generates the respective *denyaccess* action.

The Rhapsody supports event generation using RicGEN function. Events establish the asynchronous communication medium between various concurrent components. The complete features of Rhapsody state charts and the modelling can be found in [32]. The protection state transition model given in the figure-3 is used in verification of usage control implementation correctness and the other parts of state space are abstracted.

5.2 Specification of Security Properties

We use the Action-LTL language to specify the security requirements in terms of permissions granted to the users in the application over various points of time as follows

- Authors should not be allowed to review their own papers.

  ALTL: $G(permitaccess(s_1,o_1,submit)$ and $(o_1.ObjectType=Paper) \rightarrow$
  $G(permitaccess(s_2,o_2,take\text{-}reviewership) \rightarrow (s_1.ID \neq s_2.ID)$ or $(o_1.ID \neq o_2.ID))$

- The reviewers are not allowed to read other reviewer's report before submitting their own review report.

  ALTL: $G(permitaccess(s_1,o_1,submit)$ and $(o_1.ObjectType=Report)$
  $\rightarrow G(permitaccess(s_2,o_2,read)$ and $(o_2.ObjectType=Report)$ and
  $(o_1.ReportID=o_2.ReportID)$ and $(s_1.ID=s_2.ID) \rightarrow G(permitaccess(s_3,o_3,submit)$ and
  $(o_3.ObjectType=Report) \rightarrow (o_3.ReportID \neq o_2.ReportID)$ or $(s_2.ID \neq s_3.ID))))$

- After submission, no reviewer is allowed to modify his review report

  ALTL : $G(permitaccess(s_1,o_1,submit)$ and $(o_1.ObjectType=Paper) \rightarrow$
  $G(permitaccess(s_2,o_2,write) \rightarrow (s_1.ID \neq s_2.ID)$ or $(o_1.ID \neq o_2.ID))$

Similarly the other security requirements of the application can also be specified using Action-LTL.

5.3 Verification

The usage control implementation model is represented as state machines in Rhapsody software development tool suite. The usage control security requirements are specified in the dynamic verification tool's property specification file. Figure-4 shows the verification framework [27], [33] over Rhapsody. To facilitate the verification engine to sample model attributes and actions, the action handler of Rhapsody (RicGEN) is overloaded to send the verification engine a copy of every action generated after every execution step.

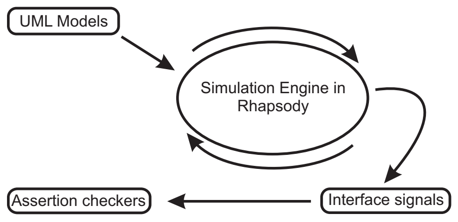

Figure-4 Formal property verification tool over Rhapsody

The verification engine which was integrated inside Rhapsody uses the local copy to verify the Action-LTL specification. The model parameters are sampled by the assertion checker separately by calling appropriate methods. The assertion monitor is built as a Rhapsody object with embedded C routines. The assertion monitor is then co-simulated with the Rhapsody model of the design-under verification and success/failures are shown as actions on the generated sequence diagram. Since, the protection state transition model is constructed using high level application actions; it can find errors in functional level usage of primitive usage control functions as well as errors in the implementation of primitive usage control functions.

6. CONCLUSIONS

This paper presented an integrated verification method for verifying the correctness of application specific usage control implementation. The verification approach succinctly captures the protection state space of the application thereby it avoids exploration of vast functional state space of the application and this method can be integrated with the industrial software development tool suites. Along with the formal specification of the usage control model and its safety analysis, the implementation verification addresses all the requirements of application specific usage control design and implementation.

ACKNOWLEDGEMENTS

The authors would like to thank Sayak Ray and Ansuman Banerjee for helping us in applying the verification tool [33].

REFERENCES

[1] X.Zhang and R.Sandhu. "Safety analysis of usage control authorization models", In *Proceedings of the 2006 ACM Symposium on Information, computer and communications security*, pp 243-254, 2006.

[2] X.Zhang, F.Parisi-Presicce, R.Sandhu, and J.Park." Formal model and policy specification of usage control", *ACM Transactions on Information and System Security,* 8(4):351-387, 2005.

[3] X.Zhang and R.Sandhu. "Safety analysis of usage control authorization models", In P*roceedings of the 2006 ACM Sympo-sium on Information, computer and communications security*, Taipei,Taiwan, pp 243 - 254, 2006.

[4] G.S.Graham and P.J.Denning. "Protection principles and practice", In *Proceedings of the AFIPS Spring Joint Computer Conference,* volume 40, pp 417-429. AFIPS Press, 1972.

[5] B.W.Lampson. "Protection", In *Proceedings of the 5th Princeton Conference on Information Sciences and Systems, Reprinted in ACM Operating Systems Review*, volume 8, pp 18-24, 1974.

[6] M.A.Harrison, W.L.Ruzzo, and J.D.Ullman. "On protection in operating systems", *Communications of ACM*, 19(8):461-471, 1976.

[7] R.S.Sandhu. "The schematic protection model:its definition and analysis for acyclic attenuating schemes", *Journal of ACM*, 35(2):404-432, 1988.

[8] V.Varadharajan and C.Calvelli. "Extending the schematic protection model - i. conditional tickets and authentication", In *Proceedings of IEEE Symposium on Security and Privacy,* Oakland, California, USA, page 213-222, 1994.

[9] V.Varadharajan and C.Calvelli. "Extending the schematic pro-tection model - ii. Revocation", In *ACM SIGOPS Operating Systems Review*, volume 31, pp 64-77, 1997.

[10] R.J.Lipton and L.Snyder. "A linear time algorithm for deciding subject security", *Journal of the ACM*, 24(3):455-464, August 1977.

[11] R. Sandhu. "The typed access matrix model", In *Proceedings of IEEE Symposium on Research in Security and Privacy*, Oakland,California,USA, pp 122-136, 1992.

[12] N.li and M.V.Tripunitara. "Security analysis in role-based access control", *ACM Transactions on Information and System Security*, 9(4):391420, 2006.

[13] P.Naldurg, S.Schwoon, S.Rajamani, and J.Lambert. "Netra:seeing through access control", In *The 4th ACM Workshop on Formal Methods in Security Engineering,* Fairfax,Virginia, pp 55-66, 2006.

[14] F.Hansen and V.Oleshchuk. "Conformance checking of rbac policy and its implementation", In *the First Information Security Practice and Experience Conference*, Singapore, pp 144-155, 2005.

[15] A. Masood, A. Ghafoor, and A. Mathur. "Scalable and effective test generation for access control systems that employ RBAC policies", In *Purdue University* Techincal *Report SERC-TR-285*, 2005.

[16] D.Senn, D.Basin, and G.Caronni. "Firewall conformance testing", In *The 17th IFIP International Conference on Testing of Communicating Systems*, Montreal,Canada, pp 226-241, 2005.

[17] A.D.Brucker, L.Brgger, and B.Wolff. "Model-based firewall conformance testing", In *8th International Workshop on Formal Approaches to Testing of Software*, Tokyo,Japan, pp 103-118, 2008.

[18] D.P.Guelev, M.Ryan, and P.Schobbens "Model checking access control policies", In *Proceedings of the Seventh Information Security Conference*, Palo Alto, U.S.A, pp 219-230, 2004.

[19] N.Zhang, D. P.Guelev, and M.Ryan. "Synthesising verified access control systems through model checking", *Journal of Computer Security,* 16(1):1-6, 2007.

[20] E. Martin and T. Xie. "A fault model and mutation testing of access control policies", In *Proceedings of the 16th ACM International conference on World Wide Web*, New York, USA, pp 667-676, 2007.

[21] Y. L. Traon, T. Mouelhi, and B. Baudry. "Testing security policies: Going beyond functional testing", In *Proceedings of the 18th IEEE International Symposium on Software Reliability,*Trollhttan, Sweden, pp 93-102, 2007.

[22] A.Pretschner, M.Hilty, D.Basin, C.Schaefer, and T.Walter. "Mechanisms for usage control", In *ACM Symposium on Information, Computer and Communications Security*, Tokyo,Japan, pp 240-244, 2008.

[23] W.Mallouli, J.Orest, A.Cavalli, N.Cuppens, and F.Cuppens. "A formal approach for testing security rules", In *12th ACM symposium on Access control models and technologies*, Sophia An tipolis,France, pp 127-132, 2007.

[24] G.J.Ahn and H. Hu." Towards realizing a formal RBAC model in real systems", In *Proceedings of the 12th ACM symposium on Access control models and technologies,* New York, USA, pp 215-224, 2007.

[25] H. Hu and G.Ahn. "Enabling verifcation and conformance test ing for access control model", In *Proceedings of the 13th ACM symposium on Access control models and technologies*, Estes Park,CO,USA, pp 195-204, 2008.

[26] A.Banerjee, K.Datta, and P.Dasgupta. "Checkspec: A tool for consistency and coverage analysis of assertion specifications", In *Proceedings of the 6th International Symposium on Automated Technology for Verification and Analysis*, Seoul,Korea, pp 228-233, 2008.

[27] A.Banerjee, S.Ray, P.Dasgupta, P. P.Chakrabarti, S.Ramesh, and P.V.V. Ganesan. "A dynamic assertion based verification platform for validation of UML designs", In *Proceedings of the 6th International Symposium on Automated Technology for Verification and Analysis,* Seoul, Korea, pp 222-227, 2008.

[28] S.Gnesi, D.Latella, and M.Massink. "Formal test-case generation for UML statecharts", In *Proceedings of the 9th IEEE International Conference on Engineering Complex Computer Systems Navigating Complexity in the e-Engineering Ag*e, Florence, Italy, pp 75-84, 2004.

[29] Y.L.Traon and J.Jezequel. "Test synthesis from UML models of distributed software", *IEEE Transactions on Software Engineering*, 33(4):252-269, 2007.

[30] www.ibm.com/software/rational. , retrieved on 12-August-2009.

[31] Object management group, "Unified language specification version 2.2" *www.omg.org/technology/documents /formal/uml.htm,* retrieved on 12-August-2009.

[32] D. Harel and H. Kugler. "The Rhapsody semantics of state charts", *In Integration of software specification techniques for applications in engineering*, pp 325-354. Springer-Verlag, 2004.

[33] A.Banerjee, S.Ray, P.Dasgupta, P. P.Chakrabarti, S.Ramesh, and P.V.V. Ganesan. "Dynamic assertion based verification platform for UML state-charts over rhapsody", In *Proceedings of IEEE region ten conferences*, Hyderabad, India, pp 1-6, 2008.