Abstract. The unified modeling language (UML) has gained wide acceptance for the design of component-based applications via diagrams (e.g., use-case, class, sequence, activity, etc.) for representing functional requirements. However, UML is lacking in its ability to model security requirements, an increasing need in today's applications. This paper presents and explains techniques that support stateful application design for secure information systems, extending the abilities of UML with role-based access control and mandatory access control. From a security-assurance perspective, we track all of the states of a design to insure that a new state (created from a prior state) is always free of security inconsistencies, with respect to the privileges of users (playing roles) against the application's components. This paper examines the theory of our approach, along with its realization as part of the software process and as incorporated into Borland's UML tool Together Control Center.

1 Introduction

In the design, development, and maintenance of secure information systems, we believe that security issues must be addressed early and often in order to potentially yield more precise and accurate security requirements. To address security during the software process, our current research [8, 9] has focused on incorporating mandatory access control (MAC) and role-based access control (RBAC) into the elements (actors, use-cases, classes, etc.) and diagrams (use-case, class, sequence, etc.) of the unified modeling language, UML [6]. In UML, while there are parallels between security and UML elements, direct support for security specification [16] is not provided. Our work [8, 9] has focused on the inclusion of RBAC and MAC by aligning the concept of role with actor, and by adding security properties to use-case, class, and sequence diagrams to capture MAC and RBAC characteristics, and lifetimes (i.e., the legal time intervals of access to UML elements). As justification for our efforts, we cite the works of [1, 2, 11, 14, 15, 20, 21] who have explored security and UML, and work in semantic and ER data models with security extensions [18, 19, 22].

Our objective in this paper focuses on capturing critical security requirements for a UML design while simultaneously tracking all of the security design states. Towards this end, this paper provides a framework that utilizes functional programming notation and constraints to capture and track all security requirements over time, resulting
in an approach to stateful UML design that can be checked against the security constraints of an information system. In the remainder of this paper: Section 2 reviews background on MAC/RBAC, and recasts our previous work [8, 9, 10] using a functional notation; Section 3 explores stateful UML design, introduces a functional model of the design state space and actions, and presents constraints for MAC, RBAC and lifetimes. Section 4 examines the safety concept, provides an example of the actions that transform the design states, and introduces design-time and post-design checking; Section 5 reviews our prototyping effort that uses Borland’s UML tool Together Control Center; Section 6 contains related work; and Section 7 concludes this paper.

2 Background Concepts

In this section, we review background concepts on RBAC/MAC, and the extension of UML integrated with RBAC/MAC security based on our prior work [8, 9, 10]. First, in MAC (or Lattice-Based Access Control [17]), security levels (SL's) forming a lattice structure are assigned to each subject (clearance - CLR) and each object (classification - CLS). The permission of the subject to perform some operation on the object depends on the relation between CLR and CLS as dictated by: Simple Security Property ("read down - no read up") [3]; Simple Integrity Property ("write down - no write up") [5]; Strict *-Property ("write equal") [17]; and Liberal *-Property ("write up - no write down") [3]. For examples, we use SLs: unclassified (U), confidential (C), secret (S), and top secret (T) where \( U < C < S < T \) (linear order) for simplicity; but our theoretical model is applicable for general partial orders. In RBAC [7, 12, 23], roles are assigned to users to specify the named assignments that those users can perform in the organization. Each role is authorized to perform some operations on certain objects.

Our extensions to UML for MAC and RBAC involve actors, use-cases, classes, and methods as realized in use-case, class, and sequence diagrams [8, 9, 10]. To accomplish this, we associate, with each UML element, an identifier, element kind, minimum and maximum security levels, and lifetime (LT) as a duration in which the usage of that element is valid. Formally, we denote \( \Lambda^\text{id} \) as the set of identification labels (uniquely assigned to each element), \( \Lambda^\text{Ek} = \{A, UC, CI, M\} \) as the set of element kinds (actor, use-case, class, and method, respectively), and \( \Lambda^\text{SL} \) as the set of security levels. Let \( T \) be a set of discrete time of the form "year-month-day [hour:minute:second]" (as a subset Cartesian product of sets of integers), a lifetime \( lt \) a time interval \([st, et]\) where \( et \) and \( st \) are the start time and end time, respectively, with \( et \geq st \). We denote the set of lifetimes as \( I \). A UML element \( x \) is a tuple \((id, k, sl_{min}, sl_{max}, lt) \in \Lambda^\text{id} \times \Lambda^\text{Ek} \times \Lambda^\text{SL} \times \Lambda^\text{LT} \times I\) where \( id, k, sl_{min}, sl_{max} \), and \( lt \) are its element identification, kind, minimum and maximum security levels, and lifetime, respectively. For a class \( c: CI \), we require \( c.sl_{min} \leq c.sl_{max} \) (since a class is a container of attributes and methods) whereas for an element \( x \) of other kinds, we require only \( x.sl_{min} = x.sl_{max} = x.sl \).

As an example, consider a Survey Institution that performs and manages public surveys. After the raw survey data is collected, senior staff adds a survey header into the database; senior or junior staff adds questions into the survey, may categorize questions, or add a question category. Questions with sensitive content are restricted to senior staff. Figure 1 depicts a use-case diagram for creating a new survey entry.
The actor ac1: Staff has two children ac2: Junior Staff and ac3: Senior Staff. Generally, Staff can perform uc2: Add Question which includes uc4: Categorize Question, and can be extended to uc5: Add Question Category if a new category must be added to the database. But, only Senior Staff can perform uc1: Add Survey Header to include a new survey header entry and uc6: Add Special Question to include special sensitive questions in a survey. Figure 2 illustrates a sequence diagram for Add Survey Header.

To illustrate, uc1: Add Survey Header with min and max SL = Secret (S) and LT = [“1/1/2005”, “12/31/2006”] is (uc1, UC, S, S, [“1/1/2005”, “12/31/2006”]). Formally, we denote Δ as the set of UML elements connected by different connection kinds, and ΛC as the set of connection kinds between UML elements x and y [8] (assuming use-case diagrams and class diagrams are acyclic) as:

- **Actor/Use-case (in use-case diagrams):** (i.1). Actor inheritance A_Ih: actor x inherits actor y; (i.2). Actor-use-case association AU_Asc: actor x interacts with use-case y by association; (i.3). Use-case inheritance U_Ih: use-case x inherits use-case y; (i.4). Use-case inclusion U_Ic: use-case x includes use-case y; (i.5). Use-case extension U_Ex: use-case y extends use-case x.

- **Class/Method (in class diagrams):** (ii.1). Class inheritance C_Ih: class x inherits class y; (ii.2). Class-method defining CM_Def: the method y is defined in class x.

- **Combinations (in sequence diagrams):** (iii.1). Use-case-class utilization UC_Uz: class y is utilized in use-case x related to a sequence diagram; (iii.2). Use-case-method utilization UM_Uz: method m is utilized in use-case x related to a sequence diagram using m; (iii.3). Actor-method utilization AM_Uz: method m is utilized by actor x; (iii.4). Method-method calling M_Ca: method x calls method y (via message passing in a sequence diagram).

To track the security assurance of UML diagrams, each connection of a source and target element (actor, use-case, class, etc.) is characterized by its connection kind and LT. Formally, a connection is a tuple (x, y, k, lt) ∈ ΔxΔxΔx×LT where x, y, k, and lt are the source and target UML elements, the connection kind, and connection LT. In
tions, the design changes to a new state that differs in SRs or UML connections. A design state is denoted by \( (\sigma^e, \sigma^c, \sigma^p) \) where \( \sigma^e \), \( \sigma^c \) and \( \sigma^p \) respectively are the state of designed UML elements, designed UML connections, and SRs.

With our approach, the design process reduces to a sequence of state transformations to add, update and/or delete UML elements, application SRs and UML connections. An action \( \alpha \) (add, update or delete) maps state \((\sigma^e, \sigma^c, \sigma^p)\) into state \((\sigma^e_{\alpha i}, \sigma^c_{\alpha i}, \sigma^p_{\alpha i})\) where the component states \( \sigma^e, \sigma^c \) and \( \sigma^p \) are represented with functions. Intuitively, \( \sigma^e \) is a function that associates each UML element and the design time with a tuple of its name, element type, security levels, and lifetime (valid at that design time). The function for the initial empty component state \( \sigma^e_0 \) returns null. \( \sigma^p \) is a function that maps a pair of UML elements, a connection kind (for a connection), and the design time to the legality (valid or invalid) of the connection. The function for the initial component state \( \sigma^p_0 \) returns null to report the absence of connections between any two UML elements. \( \sigma^c \) is a function associating a tuple of related UML elements, a SR kind (for an application’s security needs), and the design time with the enforcement needed for that SR (at that design time). If the customer requires enforcing that SR, \( \sigma^c \) is true, else \( \sigma^c \) is false (no enforcement). \( \sigma^p_0 \) for the initial component state is false to indicate that there is no SR needed by the customer.
A design state is captured with a triplet of first-class functions $\sigma^\Delta$, $\sigma^\Phi$, and $\sigma^\Theta$ and generated by an action based on the current state. Formally, the signatures of the three functions are $\sigma^\Delta : \Delta \times T \to \Lambda^{0} \times \Lambda^{0} \times \Lambda^{0} \times \Lambda^{0} \times \Lambda^{0}$; $\sigma^\Phi : \Phi \times T \to \{\text{valid, invalid, null}\}$; and $\sigma^\Theta : \Theta \times T \to \{\text{true, false}\}$. An action $\alpha$ is a meta-function that produces the triplet of first-class functions for the next state. Design reduces to an interleaving of actions to produce a state $(\sigma^{\Delta}, \sigma^{\Phi}, \sigma^{\Theta})$ and evaluate its state function $\sigma^{\Phi}$ to determine its safety.

Figure 3 illustrates the methodological concerns via the space of three design state components and their interactions at design time. Typically, the designer first creates UML elements (transforming $\sigma^\Phi$). Next, s/he associates existing UML elements with the application’s SRs (transforming $\sigma^\Theta$) which can include disallowing an actor to use some class, or specifying a static mutual exclusion of an actor to utilize two use-cases at the same time, etc. Then s/he draws connections among UML elements (transforming $\sigma^\Delta$). The legality of a connection is checked based on the security constraints (see Figure 4) of related UML elements, other existing connections, and the application’s SRs at that instant of the design time.

A brief description of the constraints for MAC, RBAC, and LTs used to validate UML connections (Figure 4) follows. Due to space limitations, only intuitive descriptions of the constraints are provided.

- **MAC Constraint.** The MAC constraint checks the domination of the SL of the source element $x$ over the target element $y$ depending on the chosen MAC properties (e.g., Simple Security). For details, please refer to our previous work [9].

- **RBAC Constraint.** Let $(a, b) \in \Phi_2$ denote the existence of a connection between UML elements $a$ and $b$ where $\Phi_2$ is a transitive binary relation representing all of the UML connections without concerning the connection kind and LT. Let $\text{connectTo}(b)$ denote the set of UML elements in the projection of the transitive closure of $\Phi_2$ on $b$ and $\text{connectFrom}(a)$ denote the set of UML elements in the projection of the transitive closure of $\Phi_2$ on $a$. The RBAC constraint for a new connection $x$ to $y$ determines whether the application’s RBAC SRs hold between $x$ and $y$ and between all the vertices $w \in \text{connectTo}(x)$ and $z \in \text{connectFrom}(y)$.

- **Lifetime Constraint.** The LT constraint for a design state at a design time $t$ requires that for any connection, $t$ fall in the non-empty overlap of the LTs of the connection and those of its source and target elements.
4 Security Assurance Design Support

In this section, we demonstrate the use of the model and constraints (see Sections 2 and 3) to obtain a safe design state. We introduce the concept of safe design state and demonstrate the way that a design state is transformed and checked against constraints when actions are performed. The concepts are foundational for a secure assurance design support program for checking the safety of states. The goal from a security assurance perspective is to assist the designer in producing a safe design. A design state \((\sigma, \delta^p, \delta^u)\) is said to be safe at design time \(t\) if every materialized connection of kind \(k\) in \(\delta^p\) between elements \(x\) and \(y\) in \(\delta^p\) evaluates to valid under SRs in \(\delta^u\) (i.e., all of the connections of the design satisfy their MAC, RBAC, and LT constraints).

When a software engineer authors the use-case diagram given in Figure 1, s/he creates new actors and use-cases and connects them with associations. Note that in our approach, the drawing of connections includes both security and functional characteristics to allow us to be able to analyze security as connections are made.

To ensure the absence of security errors, it is imperative that any actions taken by the software engineer move the application from one safe design state to the next. To illustrate this process, Table 1 provides a synopsis of the actions taken at design time \(t = “06/01/2004”\) by a software engineer who: creates new UML elements - three use-cases and three actors (States 1 to 6); adds SRs - a mutual exclusion (State 7); adds two valid connections - one between actors and one between an actor and a use-case (States 8 and 9); and fails in a connection attempt - from an actor to a use-case (State 10). The table omits the formal representation of actions, state functions, and constraints, to streamline the explanation and allow the reader to focus on the underlying concepts. When the designer attempts to draw a connection from \(ac_1\) to \(uc_1\), the connection \((ac_1, uc_1, AU_{Asc} t)\) evaluates to invalid in State 10 at design time \(t\) since the application’s SR \(ME^{SR3}\) has been added at State 7 (\(ac_2\) is prohibited from simultaneously utilizing \(uc_2\) and \(uc_3\)). Then \(ac_2\) inherits \(ac_1\) (State 9) while \(ac_1\) can utilize both \(uc_2\) and \(uc_3\) in State 9. So, \(ac_2\) can also utilize both and that violates the application’s security requirement \(ME^{SR3}\). Hence, State 10 is not safe since the evaluation of function \(\delta^{10}_{ac_1, uc_3, AU_{Asc}}\) and \(t\): \((\delta^{10}_{ac_1, uc_3, AU_{Asc}} t) = invalid\) (and the security assurance design support system reports an error message, see Section 5).

The security assurance framework supports two modes of checking: design-time for real-time checks as the designer alters a UML diagram (as presented in Table 1 and discussed above) and post-design for an on-demand check of an entire design version (iteration) across all of an application’s UML diagrams. Design-time and post-design security safety checking are realized algorithmically, as two parts of a Security Assurance Design Support (SADS) program. SADS has two sub-programs: dSADS for the design-time mode and pSADS for the post-design mode. Note that the dSADS maintains the design in the safe state by preventing the UML design tool from materializing an invalid connection (see Section 5); thus we do not need to be concerned with incremental deletion effect that would need to reconsider other connections.

To compare our model with other efforts, we make several observations. Assume that two actors, \(ac_1\) and \(ac_2\) are mutually exclusive on use-case \(uc\), and that we first connect \(ac_1\) to \(uc\). A subsequent attempt to connect \(ac_2\) to \(uc\) causes an error. Given this scenario, [13] only supports post-design (akin to our pSADS) by creating a locally
stratified logic program from the authorization specification, and as a result, there is only one connection allowed – assume that the ac1 to uc connection is chosen by rule. The approach of [4] only supports post-design, but provides all of the relevant alternatives when conflicts are identified, and the designer can either choose ac1 to uc or ac2 to uc. Our post-design conflict checking aligns closely to [4]; it reports the ME conflict which can be resolved according to the designer’s preference. Our design-time checking is also more robust since the conflict on the second attempted connection (ac2 to uc) raises the error. The main difference with [4, 13] is their use of logic programs with a fixed set of facts/rules as an authorization specification versus our use of state functions to dynamically track/adjust for changing design states.

Table 1. Example of a Design State Construction Process.

<table>
<thead>
<tr>
<th>Attempted Action</th>
<th>New State</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add uc1</td>
<td>1(σ^1,ψ^1,φ^1)</td>
<td>σ^1 adds uc1, σ^1=σ^10, φ^1=φ^0.</td>
</tr>
<tr>
<td>Add ac1</td>
<td>2(σ^2,ψ^2,φ^2)</td>
<td>σ^2 adds ac1, σ^2=σ^1, φ^2=φ^3.</td>
</tr>
<tr>
<td>Add uc2</td>
<td>3(σ^3,ψ^3,φ^3)</td>
<td>σ^3 adds uc2, σ^3=σ^2, φ^3=φ^2.</td>
</tr>
<tr>
<td>Add uc3</td>
<td>4(σ^4,ψ^4,φ^4)</td>
<td>σ^4 adds uc3, σ^4=σ^3, φ^4=φ^3.</td>
</tr>
<tr>
<td>Add ac2</td>
<td>5(σ^5,ψ^5,φ^5)</td>
<td>σ^5 adds ac2, σ^5=σ^2, φ^5=φ^4.</td>
</tr>
<tr>
<td>Add ac3</td>
<td>6(σ^6,ψ^6,φ^6)</td>
<td>σ^6 adds ac3, σ^6=σ^5, φ^6=φ^5.</td>
</tr>
<tr>
<td>Add a ME^SR to prohibit ac2 from simultaneous use of uc2 and uc1</td>
<td>7(σ^7,ψ^7,φ^7)</td>
<td>σ^7=σ^6, φ^7 with a Static Role-Objects Mutual Exclusion SR on ac2 with uc1 and uc2.</td>
</tr>
<tr>
<td>Draw a connection from ac1 to uc2</td>
<td>8(σ^8,ψ^8,φ^8)</td>
<td>σ^8=σ^7, φ^8 with an association from ac1 to uc2, φ^8=φ^5. See Note 1.</td>
</tr>
<tr>
<td>Draw a connection from ac2 to inherit ac1</td>
<td>9(σ^9,ψ^9,φ^9)</td>
<td>σ^9=σ^8, φ^9 with inheritance from ac2 to ac1, φ^9=φ^5. See Note 2.</td>
</tr>
<tr>
<td>Draw a connection from ac1 to uc3</td>
<td>10(σ^10,ψ^10,φ^10)</td>
<td>σ^10=σ^9, φ^10 with invalid connection, φ^10=φ^5.</td>
</tr>
</tbody>
</table>

**Notes:**
1. The connection (ac1, uc2, AU_Asc) is valid in State 8 (at the design time t = “06/01/2004”) as (σ^8 ac1 uc2 AU_Asc t) yields valid - all three MAC, RBAC and LT constraints hold at t.
2. The connection (ac2, ac1, A_Ih) is valid in State 9 as (σ^9 ac2 ac1 A_Ih t) yields valid - all three MAC, RBAC and LT constraints hold at t.

5 Model Architecture and Prototype Effort

Over the past year, we have been working on a prototype implementation that features a model architecture that contains a set of interacting modules (see Figure 5):

- **UML Design Tool** is the graphical user interface (GUI) for the designer to input/edit UML diagrams. Currently, we employ Borland’s Together Control Center.
with Open APIs for Java and modular plug-in structure for UML design augmented with security definition and both design-time and post-design checking.

- **Internal UML Structures Storage** stores in-core representations of UML structures extended with security properties and synchronously writes them to persistent storage (an Oracle RDBMS) in order to improve searching capabilities and support a multi-designer working environment.

- **Security Constraint Checking Module** enforces security constraints on UML elements and connections. If a constraint is violated, it pops up an error message and abandons the intended connection. Otherwise, the connection materializes in the GUI UML design tool and updates the database.

Figure 6 shows a UML property dialog with a security page (enhanced with our own custom code) that displays the clearance (CLR) for the Senior Staff actor as set to “Secret” and lifetime. Figure 7 shows a Message Pane of the plug-in module listing elements that have been created and SQL statements submitted to the RDBMS by Java. Consider Figure 8 and assume that dSADS is enabled; an attempt to connect ac1: Staff to uc1: Add Survey Header, triggers the display of an error message as the ac1:sl = C < uc1:sl = S violates the MAC constraint.
6 Related Work

There have been other efforts on security for UML and other data models. First, in [11] and [21], UML is used to represent RBAC modeling and notation. Next, in [20], UML elements are used to model MAC and RBAC based systems. Both of these efforts model security with UML, which contrasts to our approach to integrate RBAC, MAC, etc., directly into UML, allowing applications to be built with security. Third, UMLsec is an approach on formal analyses of multi-level security (secrecy and integrity) of message flows in subsystems of UML (sequence/state diagrams) and extended elements [14]. Our work on MAC is similar to theirs, and their flow analysis is akin to our checking. Fourth, in [15], SecureUML has extended meta-model elements (e.g., User, Role, and Permission) and authorization constraints expressed to generate security infrastructure for RBAC; this is similar to our constraints and checking of consistency. Fifth, [1, 2] present a framework for incorporating security into use-cases, which is more limited than our work which includes use-case, class, and sequence diagrams. Sixth, in [22], a semantic data model for security is proposed, with constraints on security levels of data entities and relationships; this work is similar to our efforts on MAC and the constraint checking as connections are made in UML diagrams. For ER models, [18, 19] have proposed extensions to represent security constraints among entities, again, similar to our efforts on MAC and constraint checking. The major difference between [18, 19, 22] and our work is that their efforts have focused on multi-level security for attributes in the relational database paradigm whereas ours is from an object-oriented approach of security levels for method level (to control the behaviors of objects) with our inclusion of RBAC and lifetimes.

7 Conclusions

This paper has proposed a formal model for design states, actions for transforming design states, and constraints on MAC, RBAC, and lifetime for UML connections. Sections 2 and 3 presented a functional model to represent UML elements, UML connections, and application security requirements, checking MAC, RBAC, and lifetime security constraints as changes are made to UML designs. Section 4 introduced the concept of safety in this context, provided an example of the material of Sections 3, and described the Security Assurance Design Support Program for design-time and post-design checking. Finally, Section 5 briefly reviewed the ongoing prototyping effort of this work into Borland’s UML tool Together Control Center, while Section 6 contrasted our work to related efforts in UML and semantic/ER data models. Our ongoing work is in a number of areas. First, as a design grows in size, it may be relevant to consider security definition and constraint checking on “meaningful” sub-sets of the design. If the UML elements within a sub-set are closely related in terms of security constraints but loosely affect elements in other sub-sets, it may be possible to compartmentalize the security analysis. Second, we are exploring additions to our constraint taxonomy, such as the Mutual Inclusion, location and dynamic constraints. Third, there is a companion research effort that is exploring the use of aspect-oriented programming to generate security enforcement code from our extended UML.
References

2. K. Alghathbar and D. Wijesekera. Consistent and Complete Access Control Policies in Use
*M74-244*, Mitre, 1975.
*Research Advances in Database and Information Systems Security*, J. Therrien (ed.), Klu-
8. T. Doan, et al. RBAC/MAC Security for UML. *Proc. of the 18th Annual IFIP WG 11.3
3rd Intl. Conf. on Computer Science, Software Engineering, Information Technology, e-
the 4th ACM Wksp. on Role-based Access Control*, 1999.
15. T. Lodderstedt, D. Basin and J. Doser. SecureUML: A UML-Based Modeling Language for
(www.omg.org.uml/).
17. S. Osborn, et al. Configuring Role-Based Access Control to Enforce Mandatory and Dis-
the 9th Intl. Wksp. on Enabling Technologies: Infrastructure for Collaborative Enterprises,
2000.
neering*, 17(11), 1991.