Integrating Access Control into UML for Secure Software Modeling and Analysis

Thuong Doan, University of Connecticut, USA
Steven Demurjian, University of Connecticut, USA
Laurent Michel, University of Connecticut, USA
Solomon Berhe, University of Connecticut, USA

ABSTRACT

Access control models are often an orthogonal activity when designing, implementing, and deploying software applications. Role-based access control (RBAC) which targets privileges based on responsibilities within an application and mandatory access control (MAC) that emphasizes the protection of information via security tags are two dominant approaches in this regard. The integration of access control into software modeling and analysis is often loose and significantly lacking, particularly when security is such a high-priority concern in applications. This article presents an approach to integrate RBAC and MAC into use-case, class, and sequence diagrams of the unified modeling language (UML), providing a cohesive approach to secure software modeling that elevates security to a first-class citizen in the process. To ensure that a UML design with security does not violate RBAC or MAC requirements, design-time analysis checks security constraints whenever a new UML element is added or an existing UML element is modified, while post-design analysis checks security constraints across the entire design for conflicts and inconsistencies. These access control extensions and security analyses have been prototyped within a UML tool.

Keywords: Constraints, Mandatory Access Control, Role-Based Access Control, Secure Software Engineering, Sensitivity Levels, UML, Use-Case and Class Diagram

INTRODUCTION

The inclusion of security in software design and development has often been an afterthought, delayed to near or post-deployment stages of the software development process or delegated to database administration. However, security has emerged as fundamental concern early and in all phases of the software process, prevalent in user interfaces (to control what each user can see), functional capabilities (to control what each user can do), and repositories (to control what data a user can access/modify). The specific focus of the article is the integration of access control into the UML (Booch et al., 1999), to provide the means for software designers and
engineers to jointly model their application’s functional and security requirements and constraints, augmented with analyses that insures the access control model characteristics, capabilities, and constraints that are being utilized are not violated. Note that despite the existence of parallels between security and elements in UML, direct support for security specification is not provided (OMG).

For access control, we leverage: role-based access control (RBAC) that focuses on user responsibilities via roles (Sandhu et al., 1996; Ting, 1988) with constraints to restrict behavior (Ferraiolo et al., 2001); and, mandatory access control (MAC) that defines classifications for objects and clearances for subjects (Bell & La Padula, 1975) with access based on the relationship between subjects and objects (Biba, 1977; Osborn et al., 2000). For security, RBAC is a flexible approach to grant/revoke permissions to/from users via roles, while MAC controls information flow (read/write on objects) for highly secure systems. Both models are augmented in this approach with lifetime constraints that determine a temporal window of activity for privileges.

This article details a practical approach that integrates RBAC and MAC into UML for secure software modeling and analysis with a two-fold emphasis. First, UML requirements definition (use case diagram) and design (class and sequence diagrams) are extended with visual and non-visual security capabilities and constraints for MAC, lifetime, and RBAC. Second, security modeling is augmented with analyses via the checking of security constraints. The design-time analysis checks these constraints as the application is created and changed as a result of every action taken by an engineering/designer. The post-design analysis (akin to a compile) checks these constraints across the entire application at a particular increment. Both the modeling and analysis capabilities have been programatically integrated into Borland’s UML design tool Together Architect (2009). The snapshots of the implementation illustrate the examples in the article. This work goes beyond our prior efforts (Doan et al., 2004a; Doan et al., 2004b) by collectively bringing all of the security extensions to UML along with their respective analyses into one context that clearly demonstrates the capabilities and potential of the work in total.

The work presented herein contrasts with other efforts on security for UML. The work of Shin and Ahn (2000) and Ray et al. (2003) simply uses UML to represent MAC and/or RBAC systems, as opposed to explicitly extending UML with RBAC and MAC. UMLsec (Jurjens, 2002a, 2002b) focuses on multi-level security (MAC) of message in sequence/state diagrams and is similar to our work on MAC extension. SecureUML (Lodderstedt et al., 2002) introduces new meta-model components and authorization constraints expressed for RBAC that involve meta-model changes. Lastly, Alghathbar and Wijesekera (2003a) incorporate security into use cases, similar to our approach. The main difference between our approach and others is one of comprehensiveness; we are doing RBAC, MAC, constraints, and lifetimes for temporal access, which combines many features of the aforementioned work with other capabilities that they do not provide.

This remainder of this article is organized as follows. Section 2 provides brief background on UML and access control models. Section 3 introduces security extensions to UML with a focus on changes to use case, class, and sequence diagrams, and support for security constraints. Section 4 presents the subsequent security analyses. Section 5, compares and contrasts our approach with other researchers’ work. Finally, Section 6 concludes the article and discusses ongoing research.

**BACKGROUND**

Section 2.1 briefly reviews UML and introduces a survey management example used throughout the remainder of the article. Section 2.2 exam-
ines role-based and mandatory access control.
Section 2.3 details the prototyping effort and platform.

**The Unified Modeling Language (UML)**

This section concentrates on use case, class, and sequence diagrams (available in versions 1.x and 2 of UML), since they are sufficient for our focused scope on security extensions and analyses of UML design (see Sections 3 and 4). A use case diagram is a collection of use cases and actors. A use case represents an encapsulation of behavior for a specific portion of an application. An actor is an external entity that interacts with software (use cases) at some level, to represent the simulation of possible events (business processes) in the system. To illustrate, consider a Survey Institution that performs and manages public surveys. Once raw data of the survey is collected, a senior staff actor will add a survey header into the database. Then, another staff (senior or junior) will add questions into that survey, and also have the ability to categorize questions and add a new question category if needed. However, special questions with more sensitive content are restricted to only senior staff. Figure 1 depicts a use case diagram for creating a new survey entry, where the actor Staff has two children Junior Staff and Senior Staff that are inherited for specialization. The actor Senior Staff has two inherited children Supervisor and Editor. Only Editor can perform the use cases: Add Survey Header to include a new survey header entry, Approve Question to accept the question content, and Verify Survey to check the whole survey. The Staff actor can perform the use case Add Question that includes the Categorize Question, and can be extended to Add Question Category if a new category must be added to the database. But, only the Senior Staff actor can perform the use case Add Special Question to include special sensitive questions in a survey. Only Supervisor can perform use case Survey Admin to decide whether the survey is ready for publishing on the Internet by extending Web Publish Survey and Activate/Deactivate Survey on Web.

A class is an abstraction for a set of objects that have the same attributes and operations. An operation is called a method; its execution against an instance of the class is called a message. Figure 2 shows the (partial) class diagram of the survey management with classes: Survey_Hdr_Add_Pg for the interface page of adding a survey_header, Survey_List for maintaining a list of surveys, Survey_Header for storing the survey header information, Question for storing question text, Special Question for storing special sensitive question text, Response for storing response data of a question, and Category for storing the category

---

**Figure 1. A use case diagram for new survey entry**

---

![Use Case Diagram for New Survey Entry]
names of a question. Survey List is a container of Survey Header. Survey Header is a container of Question. A Question is, in its own right a container of Response. A Question is associated with a Category. Special Question inherits Question and adds some restriction information.

A sequence diagram represents dynamic interactions of objects via messages, as illustrated in Figure 3 for the use case Add Survey Header. To create a new survey header Internet Usage, the Editor enters data and then clicks the submit button in the Server Header Add interface page, implemented by Survey_Hdr_Add_Pg. This action is modeled as the Editor’s sending message onSubmit to the interface Survey_Hdr_Add_Pg. On reception of the onSubmit message, the Survey_Hdr_Add_Pg object searches for the added survey title in the Survey_Repository (a Survey_List) by sending a message Survey_Title_Search to the Survey_Repository object. If a survey title does not exist in the Survey_Repository, the Survey_Hdr_Add_Pg instance sends a new header data to Survey_Repository object via the Add_Survey_Header message. On reception of the Add_Survey_Header the Survey_Repository object invokes Create_Survey_Header to instantiate a new survey header (Internet Usage) of class Survey_Header. Lastly, a new item in survey list is created via the Upd_S_List message.

Access Control Models

Our work utilizes role-based access control (RBAC) (Sandhu et al., 1996; Ting, 1988) and mandatory access control (MAC) (Bell & La Padula, 1979; Sandhu, 1993). The National Institute for Standards (NIST) RBAC model (Ferraiolo et al., 2001) lists four reference models to materialize RBAC in a secure system: RBAC0, hierarchical RBAC1, constrained RBAC2, and symmetric RBAC3. In RBAC, roles are assigned to users to specify named functions or responsibilities to be performed in the organization. Each role is then assigned a set of permissions (or privileges) that allows the role to perform specific operations on certain objects. In other words, there are many-to-many assignments of user vs. role and role vs. permission in a basic RBAC0 model. An important feature of the reference model RBAC2 is the assignments of

Figure 2. A class diagram for survey management
different constraints. For instance, the separation of duty constraints (SoD) (Clark & Wilson, 1987) serves to avoid the accumulation of too many privileges via assignment of multiple roles. Thus, the least privileges principle can be enforced.

MAC considers three abstract concepts: subject (S) which is an entity seeking to access some information; object (O) which is the information available to be provided to the subject; and, sensitivity level (SL) that refers to the extent of the confidentiality of the object or subject. The sensitivity levels in the generalized MAC form a lattice structure, but the most typical example is a linear ordering from least to most secure: Unclassified (U) < Confidential (C) < Secret (S) < Top Secret (TS). Each subject s is assigned a SL, called the clearance (CLR) that implies the capacity of s to access objects. Each object o is assigned a SL, called the classification (CLS) that specifies the protection degree for the security concern of the information provided by o. The permission of the subject s to read or write on the object o depends on the rules between the CLR of s (written CLR(s)) and the CLS of o (written CLS(o)) as defined by the following properties: Simple Security Property (read down - no read up) (Bell, 1975) where s can read o only if CLR(s) ≥ CLS(o); Liberal *-Property (write up - no write down) (Bell and La Padula, 1975; Osborn et al., 2000) where s can write o only if CLR(s) ≤ CLS(o); and, Simple Integrity Property (write down - no write up) (Biba, 1977) where s can write o only if CLR(s) ≥ CLS(o).

Prototyping Effort

For prototyping, we utilized Borland Together Architect, a Java-based product which allows the modeling of all standard UML 1.4 diagrams. The tool fully supports the synchronization of distributed diagrams, automatic code, and documentation generation. For extension purposes, the tool provides an integrated development toolkit via open application programmer interfaces (APIs) and a modular plug-in structure that easily facilitates both modifications to existing UML diagrams and the inclusion of new custom code (including graphic user interfaces, displays, etc.). The latter allows us to provide our own custom Java code. Together Architect’s structure in this regard allows us to easily include the security extensions that are changes to existing diagrams (see Sections 3.1, 3.2, and 3.4) and the creation of new information related to dependencies and constraints (see Sections 3.3 and 3.5). In addition, its GUI event

Figure 3. A sequence diagram for add survey header
handlers allow us to perform security analyses (see Section 4.3) in two situations: whenever an insertion or modification occurs on a UML diagram which prohibits the action and generates a message if there is a security violation (see Figure 10); and, for analysis across the entire design similar in concept to a compile (see Figure 11).

SECURITY EXTENSIONS TO UML

We now review the UML security extensions for RBAC, MAC, and lifetimes, focusing on use case diagrams in Section 3.1, class diagrams in Section 3.2, and sequence diagrams in Section 3.4. Section 3.3 examines the extensions for disallowed usage and mutual exclusion constraints for RBAC \(^2\), and details the UML extensions to capture dependencies between use case and class diagrams that are needed to support sequence diagram extensions and facilitate the security analysis presented in Section 4. We expect stakeholders to create use-case, class, and sequence diagrams with associated elements and connections as part of the design phase. These specifications provide access control (RBAC and MAC), associated constraints, and information on when access is allowed (referred to as lifetimes). To facilitate the analysis, we further assume that the connections in use case, class, and sequence diagrams are acyclic, and that multiple inheritance (in use case diagrams among use cases and actors and in class diagrams among classes) is not allowed. The analysis insures that all access control model characteristics, capabilities, and constraints are never violated as a UML application is created and modified.

Use Case Diagram Extension

Use-case diagrams have been extended by: the alignment of actors to roles, the inclusion of MAC sensitivity levels (see Section 2.2) for use cases and actors, and the use of lifetimes to track temporal limits on privileges. First, by aligning actors to roles we establish a link from an actor to a role in NIST RBAC \(^1\), (Ferraiolo et al., 2001), and we align actor inheritance with the comparable role inheritance of RBAC \(^2\). From a security privilege perspective, each of the lines that connect an actor to a use case represents an assigned privilege. In Figure 1, Supervisor is assigned to the use cases Survey Admin and Web Publish Survey, and also inherits the privileges associated with the Senior Staff (parent) and Staff (grandparent); this acquisition is consistent with the NIST model. There is also the implicit assignment of other connected use cases, linked to directly assigned use case by include, extend, or inheritance. Collectively, all the directly or indirectly reachable use cases represent the assigned privileges on an actor-by-actor basis.

Second, we extend the use case diagram with MAC sensitivity levels and security properties (see Section 2.2). For each actor or use case in a use case diagram we define two clearances or classifications, respectively, as related to its read and write capabilities. Recall that security properties (Strict *, Simple Security, etc.) differentiate between read and write access for SLs. To be consistent in realizing SLs in UML, we differentiate between read and write access for CLR of actors and CLS of use cases. In fact, since actors are connected to use cases, and use cases are connected to one another, assigning a single CLR or CLS is not feasible. Instead, for each actor or use case, we assign a read security capability (RSC) with a minimum value (RSC.min) and a maximum value (RSC.max), and a write security capability (WSC) with a minimum value (WSC.min) and a maximum value (WSC.max); each set of values represents a range of SLs. The RSC/WSC refers to the capability of the UML element to obtain or modify the application’s information (stored in the attributes of some classes) via an interaction chain with other UML elements (via UML connections) requiring that the SL of the accessed information lies within the RSC/WSC of the accessing UML element. The minimum and maximum SLs of RSC/WSC are taken from a set, such as \{Unclassified, Confidential, Secret, Top

---

Copyright © 2010, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.
Secret). Figure 4 is an illustration of the UML inspector window showing the RSC/WSC for the actors Staff and Editor corresponding to the use case diagram of Figure 1. These UML inspector windows from Borland’s Together Architect have been extended (using available APIs) to include the Security Properties tab. In the Figure, the Read Min (Max) Value refers to RSC.min (RSC.max) and the Write Min (Max) Value refers to WSC.min (WSC.max). In Figure 4, actor Staff has RSC.min of Unclassified and RSC.max of Confidential and WSC.min of Unclassified and WSC.max of Confidential, while actor Editor has Read Values of Unclassified and Top Secret and Write Values of Confidential and Secret. The Read and Write values of Add Question are also shown.

The third addition is the concept of a lifetime (LT), a temporal range that indicates when the particular UML element is active from a security perspective. Outside of that period (before or after), the role is not available to be used within the application. As an abstract concept, a lifetime can be a continuous time interval, a repeating interval (e.g., one day per week), or any quantifiable collection of intervals. For simplicity, we assume a temporal interval of the form [starting time, ending time]. As shown in Figure 4, the Start and End Dates are given for the two actors and one use case. These are clearly coarse grained in this prototype (day-month-year); yet, from a modeling perspective, these can be very fine-grained. Further, we extend UML connections by adding an LT to the connection between actor and use case, and include/extend associations, to indicate when the interaction (from actor to use case or between use cases) is allowed. When a designer draws a connection from the actor Staff to the use case Add Question (see Figure 1 for the connection), the designer is prompted for its LT. We do not apply lifetimes to inheritance since this is a condition that always holds, i.e., it is not possible for an ancestor to exist only some of the time. One final note, the alignment of roles to actors, the linking of connections to direct and implicit privileges, the inclusion of MAC sensitivity levels, and lifetimes for allowable access time periods are leveraged so that when an addition or change is made to a diagram, it is automatically analyzed. For example, drawing a new connection between an actor and a use case may be disallowed if the sensitivity level of the actor does not appropriately dominate the sensitivity level of the use case. The security analysis will be discussed in Section 4.

Class Diagram Extension

The extensions to class diagrams includes both the Read/Write Security Capacities (RSC/WSC) and a lifetime for every class and each method of a class, with LTs tracked for class association and class aggregation; we treat inheritance as a condition that always holds regardless of time since a class can’t have an ancestor only some of the time. To illustrate, in Figure 5, we excerpt from the class diagram of Figure 2 to show the Survey List class and two inspector windows. Survey List has RSC.min of Unclassified and RSC.max of Top Secret, WSC.min of Confidential and WSC.max of Secret, and LT of 01/01/2005 to 12/31/2010. The method Add_Survey_Header has RSC.min of Unclassified and RSC.max of Top Secret, WSC.min of Confidential and WSC.max of Secret, and LT of 01/01/2005 to 12/31/2010. Note that the RSC/WSC for the Survey List must subsume the RSC/WSC values for every one of its methods; once the class value is set, the RSC/WSC values of the methods are constrained when a designer creates a new method or modifies the RSC/WSC value of an existing method. This is also true for LTs, where a method can’t have a LT outside the LT of its class. Further, the LT of a parent must subsume the LT of all of its descendants both direct and indirect. For inheritance, there is also a subsumption among RSC/WSC sensitivity levels. Lastly, whenever classes are connected by association
or aggregation, lifetimes are tracked for those connections, and must be consistent with the source and destination classes being connected (not shown).

**Connections**

UML does an excellent job of tracking associations in its structural and behavioral diagrams. However, there is no diagram that can show directly what structural elements (classes) should be used for realizing which behavior elements (use cases). Recall that when an actor is connected to a use case (either directly or indirectly), the privileges of that actor/role are defined (see Section 3.1). Use cases are functional entities to be eventually implemented using classes (and their methods). For security consistency tracking, it is imperative to associate use cases to the classes and methods that realize them to create a chain where privileges can be traced from a role (actor) to its use cases and ultimately to classes and their methods. Thus, we introduce connections from a use case to a set of classes to be utilized to realize the use case, and for each class in the set, to identify a subset of its methods. The first new connection, *Use Case-Class Utilization*, allows the designer to assign a subset of the set of classes to a use case, shown in the top of Figure 6 for the *Add_Survey_Header* use case. This is a new window that has been coded and included as part of Borland’s Together Architect. For security analysis, the Use Case-Class Utilization case is checked on the RSC/WSC and lifetime properties of the use case and the class to guarantee the MAC and LT security requirements hold. The second connection, *Use Case-Method Utilization*, allows the designer to assign the methods of each selected classes to be utilized in realizing the use case. As shown in the bottom of Figure 6, for a given use case, the Utilized Class can be selected, displaying all of its methods, from which a subset can be selected. For security analysis, the Use Case-
Method Utilization is checked on the RSC/WSC and LT properties of the use case and the method to guarantee the MAC and LT security requirements hold.

**Sequence Diagram Extension**

In a sequence diagram, the actors and methods already have the RSC/WSC and LTs. To represent the message passing, the designer may draw: *Actor-Method Utilization* from an actor to a method (callee) and *Method-Method Calling* from a method (caller) to another method (callee); each of these two new connections is augmented with a LT to indicate the temporal condition that allows the actor or caller method to utilize the callee method. Figure 7 shows the MAC and LTs of actor *Editor*, and methods *OnSubmit* and *Survey_Title_Search*. The designer must also provide LTs of the connection from actor *Editor* to method *OnSubmit* and from method *OnSubmit* to method *Survey_Title_Search* (not shown). Note that the LTs of the Actor-Method Utilization or Method-Method Calling limit the time span of usage from a security perspective.

**RBAC Constraints**

To complete our extensions to UML, we discuss the inclusion of RBAC constraints that are realized as part of the NIST RBAC standard (Ferraiolo et al., 2001). RBAC constraints can be defined in different ways. First, *disallowed usage constraints* are utilized to identify essentially negative privileges, defining what use cases, classes, and methods that an actor (role) is prohibited from using. Second, *mutual exclusion constraints* restrict the role-permission assignment to define the conditions under which one role uses multiple UML elements (use case, classes, and methods) at the same time or two or more roles are allowed to access (or not) the same UML element. A *disallowed usage constraint* is a negative permission, and is specified to explicitly prohibit an actor (role) from using some non-actor UML element (use case, class, or method). A disallowed usage as a negative permission is necessary to explicitly prevent the indirect (positive) permission of an actor (for instance, via the inheritance of its parent actor) to utilize some non-actor element. Continuing with the example from Section 2, suppose that the security policy required by the
customer states that the actor Junior Staff must not utilize the use case Add Special Question which is intended for other role(s). The designer specifies disallowed usage (DisU) as shown on the left side of Figure 8 via a new window that extends Borland’s Together Architect.

The remaining two constraints are related to mutual exclusion as a mean for separation of duty that helps to enforce the least privileges principle by limiting the accumulated privileges via multiple user-role assignments. Note, we only consider the static mutual exclusion constraints between actors (roles) and non-actor UML elements. First, for a static object-roles mutual exclusion constraint (ME(SOR)), an object (a non-actor UML element) is not allowed to be utilized by multiple actors at the same time based on the enterprise’s separation of duty requirements. This means that two or more actors are prohibited from using the same use case, class, or method at the same time; there can be multiple ME(SOR) constraints. For example, if the security policy required by the customer states that the actors Supervisor and Editor must not utilize the use case Activate/Deactivate Survey on Web at the same time. The designer specifies a ME(SOR) as shown in the right side of Figure 8.

Conversely, a static role-objects mutual exclusion constraint (ME(SRO)) prohibits an actor from using multiple non-actor UML elements at the same time based on the enterprise’s separation of duty requirements. In other words, to support the “check and balance” principle or to avoid the conflict of interest, the security officer may create a ME(SRO) constraint to disallow an actor to utilize a set of non-actor UML elements at the same time. For example, suppose that the security policy required by the customer states that the Junior Staff actor must not utilize both the Add Question and Approve Question use cases (not shown). Both types of static mutual exclusion constraints are defined at design time and enforced at runtime. As before, the definition of disallowed usage and static mutual exclusion introduces another area for analysis to make sure that these definitions and other security and non-security features are conflict-free as the design is created and modified.

Figure 7. An extended sequence diagram for add survey header
SECURITY ANALYSES

This section defines the conditions for secure design analyses in UML involving RBAC, MAC, and lifetime extensions to use-case, class, and sequence diagrams, as well as RBAC constraints for disallowed usage and mutual exclusion (see Section 3). Section 4.1, discusses the underlying mechanism that facilitates analysis, which involves the treatment of a UML design as a connected acyclic graph and the computation of the closure(s) of a UML element (or connection) to determine sub-graphs of reachable elements (or connections). Then, Section 4.2 enumerates all of the available analyses including: MAC sensitivity level analyses both within and between UML elements, lifetime analyses; and, RBAC constraint analyses for disallowed usage and mutual exclusion. Section 4.3 then discusses when the analyses can occur: design-time analysis institutes checks whenever a designer creates a new UML element/connection or modifies an existing element/connection; and, post-design analysis, akin to a “compile”, institutes a complete analysis of the entire design at a design increment or milestone.

Mechanism for Facilitating Analysis

A UML design that has been created with security as described in Section 3, is essentially a directed acyclic graph with connections from actors (roles) all the way through to methods. The connections in a UML design include: actor-to-actor inheritance; use case inheritance, include, and extend; actor-to-use-case utilization; use-case-to-class and use-case-to-method realization (via new use-case to class and method connections discussed in Section 3.3 and shown in Figure 6); class inheritance, aggregation, and association; and method-to-method linkage (once there is code that implements each method). These connections are shown in Figure 9: Actor1 inherits from Actor2 which utilizes UseCase1 and UseCase2; UseCase1 inherits from UseCase3; UseCase2 includes UseCase4 which is extended with UseCase5; UseCase4 utilizes Class1 and Class2 for its realization, including two methods from Class1 (Method11 and Method12) and one from Class2 (Method21); and, Actor1 interacts directly with UseCase6. Using this diagram, it is possible to compute a forward closure of elements (FCE) to identify all directly or indirectly reachable UML elements (actors, use cases, classes, methods) connected to a given UML element: FCE(Actor1) = {Actor1, Actor2, UseCase3, UseCase5, UseCase6, UseCase7, Class1, Class2, Method11, Method12, Method21}. Note that we have not included method-to-method connections, but if available, these would be part of the closure. A forward closure of connections (FCC) can be computed: FCC(Actor1) = {Conn1…Conn12} with Conn13 and Conn14 are not included. A backward closure of elements (BCE) is the set of all UML elements that are
connected directly or indirectly via incoming connections: BCE(\text{UseCase5}) = \{\text{Actor1, Actor2, Actor3, UseCase2, UseCase4}\}. All of these closures are important for the analysis, since they provide the means to identify from a given UML element, all other linked elements and their connections. The consistency analysis for the lifetimes (discussed in Section 4.2) requires both the FCE and BCE in order to examine their overlap (intersection).

Enumerating Security Analysis Capabilities

Security analysis involves the checking of consistency of UML elements and connections in regards to MAC sensitivity levels (read and write SLs) and associated properties (e.g., Simple Security, Simple Integrity, Liberal-*), etc., in Section 2.2), lifetimes, and RBAC constraints on disallowed usage (UML elements a role cannot access) and mutual exclusion (multiple roles against the same UML element or one role against multiple elements). This section reviews each of these three types of checks in turn. To begin, recall that LTs are assigned to each UML element, and to the connections between each element (see Section 3 and Figures 4, 5, and 6). Conceptually, when two UML elements are connected with one another (e.g., an actor to a use case) which are also not connected to any other UML element, the intersection of the LTs at the source and destination of the connection are checked for overlap; if so, the connection is allowed. In general, the lifetime analysis operates as follows:

- **Upon the insertion of a new UML connection:** For a UML source element \(se\) and destination element \(de\) of the connection \(c\), calculate the closure set \(CS = \{\text{BCE}(se), \text{FCE}(se), \text{FCE}(de), \text{FCC}(c)\}\). This determines all reachable elements and involved connections. The intersection of all lifetimes for all elements in \(CS\) is calculated and if there is a non-null overlap, then the connection is allowed; otherwise, the connection is prohibited.
- **Upon the updating of the lifetime of an existing UML element or connection:** For a given UML element \(e\), calculate \(CS = \{\text{BCE}(e), \text{FCE}(e), \text{FCC}(c)\}\) and check the intersection; if non-null, allow the lifetime change, otherwise do not.

In practice, as a designer creates a UML model over time, adding elements and connections, the analysis of LTs constantly occurs, so this design time analysis is very easy, often involving just the source and destination (since the elements in the closure have already been
checked during earlier connections). However, as discussed in Section 4.3, a designer can turn off the automatic analysis and then analyze the entire design at a particular increment or milestone (much like a compiler). In this case, the closure sets must be calculated in total for each UML element to analyze the lifetimes across the entire design at once (rather than one connection at a time).

The second security analysis involves MAC with respect to sensitivity levels (SLs) and properties of UML elements. There are two types of analyses. First, \textit{intra-element SL analysis} involves a limited examination of the read and write security capabilities (RSC/WSC – see Section 3.2) for each UML element to make sure that the RSC.min ≤ RSC.max and WSC.min ≤ WSC.max, meaning that the minimum clearance (or classification) cannot exceed the maximum. Intra-SL analysis is performed when a designer creates a New UML Element (so that the RCS/WSC min and max are defined) or updates an Existing UML Element’s min or max. Next, \textit{inter-element security property analysis} involves the checking of the read and write, min and max, between two elements, based on the security properties that have been chosen. Recall from Section 2.2 the four properties Simple Security (read down - no read up), Strict *- (write equal), Liberal *-, and Simple Integrity (write down - no write up), which involved comparing sensitivity levels (Unclassified, Confidential, Secret, and Top Secret) with one another (e.g., for Simple Security, CLR(s) ≥ CLS(o), and for Liberal *- CLR(s) ≤ CLS(o)). For the SLs of two UML elements, based on the chosen security properties (one for read and one for write), the RSC/WSC mins and maxes of the source and destination must be compared. Again, there are two cases for \textit{inter-element security property analysis}:

- **Upon the insertion of a new UML connection c**: Calculate the closure set \(CS = \{FCC(c)\}\) and for each connection \((se, de)\) in \(CS\) check \(se.RSC.min\) against \(de.RSC.min\), and \(se.RSC.max\) against \(de.RSC.max\). Compare the sensitivity levels according to the defined read (e.g., Simple Security) and write (e.g., Liberal-*) properties. All four comparisons must succeed for all connections in \(CS\), for the connection to be allowed.

- **Upon updating the sensitivity level of an existing UML element c**: Calculate \(CS = \{FCC(c)\}\) and compare RSC/WSC min/max for source/destination based on chosen read/write security properties.

For design-time analysis, the \textit{inter-element security property analysis} is quick, since only the single connection or directly connected elements must be checked (earlier connections or changes must have previously been checked); for post-design analysis, the computation of \(CS\) for each connection across the entire design is required, with the comparison of SLs/properties as described previously.

The final security analysis involves RBAC constraints for disallowed usage and mutual exclusion. Recall that disallowed usage (see Section 3.5 and left side of Figure 8) allows a designer to identify, for each actor (role), a list of prohibited use cases, classes, and/or methods, which results in the creation of the disallowed usage set (DUS) for each actor (which may be null). For a given actor \(ac\), the \textit{disallowed usage analysis} calculates \(\{FCE(ac)\} ∩ DUS(ac)\); a non-null result means there is a conflict between disallowed privileges and assigned privileges.

For mutual exclusion, recall that there are two cases: \textit{static object-roles mutual exclusion constraint (ME(SOR))} which prohibits two or more actors (roles) from using the same UML element at the same time; and, \textit{static role-objects mutual exclusion constraint (ME(SRO))} which prohibits an actor (role) from using two or more UML elements at the same time. For the ME(SOR) analysis, a prohibited actors set (PAS) is maintained for each UML element. For any given element \(e\) (use case, class, or method), the analysis calculates \(\{BCE(e)\} ∩ PAS(e)\); if
the intersection is not empty, there is a conflict. For the ME(SRO) analysis, a prohibited elements set (PES) is maintained for each actor. For any given actor ac, the analysis calculates \( \{FCE(ac)\} \cap PES(ac) \); if the intersection is not empty, there is a conflict. The check for all RBAC constraints is performed when an existing actor is connected to a use case (or a parent actor), a new connection that impacts the actor is added, a new disallowed usage element is added to the actor, or a mutual exclusion constraint involving the actor is modified.

**Design-Time and Post-Design Analyses**

The six security analyses (lifetime analysis, intra-element SL analysis, inter-element security property analysis, disallowed usage analysis, ME(SOR) analysis, and ME(SRO) analysis) are utilized in two different ways. First, design-time analysis occurs throughout the design process, with the analyses embedded into the UML tool (Borland Together Architect) so that whenever an action is performed (e.g., create a UML element, connect two existing elements, update an existing element, modify an RBAC constraint, etc.), the appropriate check occurs as detailed in Section 4.2. Second, we recognize that attempting to keep a complex design in a security consistent state during the design activity may be impossible; there are different class diagrams (some partial) being developed by different individuals and when users starts connecting them to one another (bring together the design components), the potential for conflicts is extremely high. As a result, designers can turn-off design-time analysis in the UML tool, and create a design with security extensions (MAC, lifetimes, RBAC constraints, etc.). At any design milestone, post-design analysis, akin to a compile, can be performed to analyze the entire design (all UML elements and connections) and report a list of errors. Both analyses are discussed below.

*Design-time analysis* runs in the background as the designer creates or alters UML diagrams for an application; the six analyses of Section 4.2 are always “ready-to-run” with hooks inserted into the UML tool to automatically trigger the appropriate analysis based on the user’s action. Figure 10 illustrates this process where the designer attempts to connect the use case *Add Survey Header* with the Actor *Staff*, MAC inter-element security property

---

*Figure 10. Inter-element security property analysis*
analysis determines a conflict in the clearance (CLR) of the source (actor Staff has as sensitivity level of C) with the classification (CLS) of the destination (use case Add Survey Header has a sensitivity level of Secret). This conflict may result from the read or write security capability, and therefore the connection is not allowed to occur. Design-time analysis can be considered as taking the state of a UML diagram, evaluating a potential future state (with the change), and allowing a transition to a new state if there is no conflict. From a security perspective, there is a series of states based on the action taken by the designer, with security analysis performed based on the type of action as discussed in Section 4.2. In this way, the system always maintains a consistent security state. Note that we presented and proved a theorem that guarantees that, as long as the design-time analysis runs at all time, a user will produce design revisions that are always consistent (Doan, 2008).

Post-design analysis is utilized when a design becomes sufficiently large (containing numerous UML elements, connections, and RBAC constraints) and complex (bringing together different design components/diagrams) that design-time analysis would always find conflicts, effectively stalling the design process. Consequently, we provided in the prototype integrated into Borland’s Together Architect the ability to turn off the design-time analysis. Once off, the designer can create a UML design with elements, connections, security features, etc., without continuous analyses. At regular milestones, the post-design analysis can be executed to produce an overall check (all six of the analyses in Section 4.2) of the design by iterating through all elements and all connections; this is shown in Figure 11. Since a connected UML design is an acyclic graph, the complexity of post-design time analysis is based on the number of UML elements (nodes) and connections (edges), coupled with the number of RBAC disallowed usage and mutual exclusion constraints. In the worst case, this is $O(m^3)$ where $m$ is the maximum of the number of nodes, edges, or RBAC constraints.

Figure 11. Post-design analysis
RELATED WORK

In this section, we compare and contrast our work with the approach of others. In Epstein and Sandhu (1999), a proposed Framework for Network Enterprise utilizes UML notations to describe a role-based access control model, employing UML as a language to represent RBAC requirements. However, the representation is too general to incorporate subtle properties of RBAC such as separation of duty constraints (Ferraiolo et al., 2001) for different types of RBAC constraints. Shin and Ahn (2000) proposed an alternative technique to utilize UML notations to describe RBAC modeling and processing. With a more general approach, Lodderstedt et al. (2002) proposed SecureUML as an UML-based modeling language for RBAC. Their work introduced new meta-model components (e.g., User, Role, Permission, ActionType, ResourceType etc.) based on the UML stereotype definition (for the abstract level). These meta-model stereotypes express, for instance, the RBAC requirements among users, roles, permissions, types of actions that the permission can perform, and the resources that are affected by the action, etc. Note that their effort has been extended in (Basin et al., 2006) using model-driven architectures to assist in the security definition process via model transformation. The common objective of all of these efforts was to focus on the way that UML elements can be used to model roles rather than taking a larger view of examining secure software design with state tracking and constraint checking, which has been our focus.

In an effort similar to our work, Jurjens (2002a, 2002b) proposed UMLsec with extended UML features to accommodate security requirements. Their effort utilizes a mathematical Abstract State Machine model to formalize UML elements (no use cases) and extend several stereotypes to accommodate their proposed security framework towards theoretical security verification with UML. This is in contrast with our approach to extend properties of essential UML elements (use cases, actors, classes, and methods) and keep track of the evolving design state using functional representation in order to directly apply security models for secure software design.

In another effort, Alghathbar and Wijesekera (2003b), introduced a framework called AuthUML for incorporating security into use cases, with the assumption that the designer can specify a list of conceptual operations and their affected objects in each use-case. The authors indicate that their work is an application of the Flexible Authorization Framework (Jajodia et al., 2001) using Prolog-like predicates to represent RBAC constraints. The predicates are built in a special order to be shaped into a stratified logic program (for obtaining a stable model). Their approach is different from ours in terms of computational representation.

Our approach extends the UML with a RBAC model that is roughly at the constrained RBAC level in the NIST RBAC model (Ferraiolo et al., 2001). The symmetric RBAC model is not taken into account in our RBAC UML extension since the UML model does not consider specific users; UML only deals with actors as roles in RBAC. On the aspect of temporal security, Bertino et al. (2000) proposed a hierarchical temporal authorization model where the time span can be represented as periodic intervals. In a security model for distributed component-based environment, Phillips et al., 2002a; Phillips et al., 2002b) represented the lifetimes as the fixed intervals with the format of starting time/date and ending time/date. Our lifetime representation is more general since it is built on the concept of lattice and a partial order relation, and can represent many kinds of lifetime such as fixed intervals. Overall, we believe that these and other efforts were helpful resources for our research goal that targets the integration of two access control models into UML.

Finally, there have been efforts related to security analysis. UMLsec as discussed above has been extended with security analysis in Juergens et al. (2008) in which their UML security specification is translated into Prolog and an automated theorem prover analyzes it for correctness. Both their and our effort
performs static analysis at design time. The recent work of Basin et al. (2009) extends their SecureUML work (see above) so that modeled security properties can be evaluated against hypothetical run-time instances; this includes static analysis regarding users and permissions to roles which is similar to our work, but goes beyond our effort to consider the implications at runtime. Another effort of note in Sohr et al. (2008) works towards systematically verifying and validating non-temporal and history-based authorization constraints via UML’s Object Constraint Language (OCL), which are then verified using a theorem prover. Such an approach could be utilized to underlie our current analysis for RBAC disallowed usage and mutual exclusion constraints.

CONCLUSION

This article has presented the integration of role-based access control (RBAC), mandatory access control (MAC), lifetime, and RBAC constraints into UML to provide a security model and associated security analyses. Our approach and the checking mechanism have been integrated into a UML tool to check and maintain design states as an application’s design is created and modified over time. Section 2 presented background material on UML and RBAC and MAC. Using this as a basis, Section 3 presented the proposed extensions to UML, including: the treatment of actors as roles and their connections to use cases as positive privileges (Section 3.1); extensions to class and sequence diagrams for sensitivity levels and lifetimes (Sections 3.2 and 3.4); the introduction of new connections for use cases to classes and methods (Section 3.3); and, disallowed usage and mutual exclusion constraints (Section 3.5). For all, a common example was expanded upon, with screen shots that illustrated the changes and additions to the Borland UML Tool Together Architect. This naturally set the context for the analysis capabilities in Section 4, which included the concept of a closure for UML elements and connections (Section 4.1), the six different analyses for lifetimes, MAC, and RBAC constraints (Section 4.2), and design-time and post-design analyses (Section 4.3).

In terms of ongoing and potential future work, the areas include: integration of the lifetime into the RBAC constraints; optimization of security analysis algorithms; extension to include discretionary access control (Sandhu & Munawer, 1998); expansion to consider other UML diagrams; and, transitioning to implementation via secure code generation. Of particular note is the last item on that list, which may allow us to align with other research on extending UML for secure software engineering (Pavlich et al., 2008). In this complementary effort, the focus has been on proposing new security UML diagrams for capturing privileges based for users, roles, and delegation of authority (discretionary access control), which is then coupled with the automatic generation of aspect-oriented security enforcement code. The motivation for separate diagrams is to collect security that is distributed across the design into named diagrams; this complements our motivation to add security into UML with a limited impact by tweaking existing diagrams and their properties. There is a natural fit between the two efforts, since the definition of security in our model (Section 3) can be used to generate the new security UML diagrams of their work (and vice versa), which would allow our design extensions to automatically transition into enforcement code.

REFERENCES


Thuong Doan is a researcher at Northside, Inc., which conducts research on natural language processing and applies said artificial intelligence to computer gaming. He completed his PhD in computer science & engineering at the University of Connecticut with an emphasis on a framework for software security in UML with assurance. Doan has published 1 journal article, 3 book chapters, and 6 refereed conference/workshop articles.

Steven Demurjian is a full professor of computer science & engineering at the University of Connecticut with research interests of: secure-software engineering, security for collaborative Web portals with biomedical applications, and security-Web architectures. Demurjian has over 130 archival publications, in the following categories: 1 book, 2 edited collections, 41 journal articles and book chapters, and 87 refereed conference/workshop articles.

Laurent Michel is an associate professor of computer science & engineering at the University of Connecticut with research interests of: combinatorial optimization, constraint programming, local search, programming languages, and artificial intelligence. Michel has over 65 archival publications, in the following categories: 2 books, 22 journal articles and book chapters, and 41 refereed conference/workshop articles.

Solomon Berhe is a doctoral student in computer science & engineering at the University of Connecticut working with Demurjian. His current research focuses mainly on fine-grained role-based access control (RBAC) in Wikis. Berhe’s most recent research interests include investigating collaborative access control issues in the health care domain. Berhe has published 1 journal article, 2 book chapters, and 3 refereed conference/workshop articles.

Copyright © 2010, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.