UML Design with Security Integration as First Class Citizen
Thuong Doan, Steven Demurjian, Reda Ammar, and T.C. Ting
Department of Computer Science and Engineering,
University of Connecticut
371 Fairfield Road, Storrs, CT 06269-2155, USA
E-mail: [thuong,steve,reda,ting]@engr.uconn.edu

Abstract
Security for software applications involves defining what needs to be protected (security policy), authorizing privileges of the application to users, authenticating application users, and providing a high degree of security assurance in regards to the access of users to the application. To address security during software design/development, our previous work has proposed a model to incorporate role-based access control (RBAC) and mandatory access control (MAC) into the unified modeling language (UML) to support the definition of security for software applications. Included in this work is a series of design-time checks that insure that the defined RBAC/MAC security is always consistent as a UML design with security properties is created and modified. In this paper, we extend this effort by proposing a formal model that combines typed logic with active database language concepts in order to support the checking of constraints during design-time (as UML diagrams are created and modified) and post-design (for the entire UML design that represents a version) towards the attainment of security assurance. To demonstrate the feasibility and utility of our work on secure software design, our RBAC/MAC enhancements and the constraint checking has been integrated into Borland's UML tool Together Control Center.

Keywords: RBAC, MAC, UML, access control, secure software design.

1. Introduction
The scope of security for the design, development, and maintenance of software applications is wide ranging, and includes: security policy definition for what needs to be protected, the anticipated users, user privileges, etc.; authorization to grant/revoke privileges to users based on the policy; authentication to verify the users and limit their actions to authorized privileges; and, security assurance to attain the security policy within the enforcement framework for an active application. As a consequence, the embedding of security into software applications for enterprises is no longer an option, but has become a critical requirement. In today's world, it is unacceptable to delay or defer security to latter stages of the process or to delegate its responsibility to database administration. Instead, security must be brought to the forefront as a first class citizen at early and all stages of the software design and development process.

To address security during software design and development, our research [12, 13] has focused on incorporating role-based access control (RBAC) and mandatory access control (MAC) into the unified modeling language, UML [9]. In UML, different diagrams capture an application's behaviors, interactions, and implementation for stakeholders (e.g., users, designers, software engineers, etc.). While there are parallels between security and UML elements, direct support for security specification in UML [19] is not provided. There have been other efforts on security and UML: [14] and [24] used UML as a language to represent RBAC modeling and notation; [22] used UML elements to model MAC and RBAC based systems; [17] on theoretical security definition with UML; [18] introduced SecureUML for model-driven security with extended meta-model elements for RBAC only; and, [1, 2] presented a framework to incorporate security into use-cases only. Our own work [12, 13] has focused on the inclusion of RBAC and MAC in a manner consistent with UML by aligning the concept of role with actor, and by adding properties to use-case, class, and sequence diagrams to capture MAC and RBAC. This paper extends this work with a formal model for constraint checking, based on formal typed logic [25] combined with [4] for active databases.

Our objective in this paper is on formalizing constraint checking for design-time (as changes to UML diagrams are made) and post-design (at regular milestone/versions) security assurance for RBAC/MAC modeling in UML [12, 13] utilizing a three-stage methodology for secured software design:

- **Stage 1:** Create and refine UML diagrams for an application, augmented with security features, according to the software specification which includes its security requirements.

- **Stage 2:** Verify the security constraints on the UML diagrams. The UML diagrams are modified as in Stage 1 until all of the security constraints defined on the design (i.e., UML diagrams) hold.

- **Stage 3** (our future work): Validate the security implication derived from the UML diagrams with security consistency in Stage 2 against the security requirements from the customer's (formal or even informal) specifications.

In support of Stage 1, we have extended the UML by integrating RBAC/MAC features into the UML elements and diagrams [12, 13]. Briefly, we have augmented UML elements (use-cases, actors, classes, and methods) with extended properties like security levels (for MAC), lifetimes (availability of elements from a security perspective), and assigning business roles to actors (for RBAC). To supplement this
definitional support, we provide MAC and RBAC constraints on relationships between UML elements in diagrams. The MAC constraints generally require the domination of security level between elements, e.g., the clearance of the actor's role must dominate the classification of the use-case employed by that actor in order for the actor to be connected by a relationship to the use-case. For RBAC constraints, we added role privilege relations on actors with non-actor elements such as use-cases, classes, and methods to specify the actor's (role's) privilege of utilizing the non-actor elements during a lifetime (constraint). Overall, the intent of RBAC and MAC constraints is to ensure that the defined UML elements (with security) are consistent with security properties and constraints.

The goal of Stage 2 is to verify that the MAC and RBAC constraints for the UML diagrams in Stage 1 are satisfied without inconsistency, which is the emphasis of this paper. Specifically, we formalize the checking of RBAC/MAC constraints in UML diagrams described in our previous work [12, 13] by utilizing typed logic combined with a subset of the active database language $L_{active}$ [4]. With this as a basis, we propose two modes of checking: design-time (instantly check whenever the designer draws a new connection or deletes an existing connection in a UML diagram for an application) and post-design (run the checks across all connections after the designer has reached a milestone or version in an application's UML diagrams). Both checking modes are performed by algorithms that are part of a Security Assurance Design Support (SADS) program component that is designed to be integrated into a UML design tool, which for our purposes is Borland's Together Control Center (TCC).

In the remainder of this paper: Section 2 reviews background concepts of our model in previous work [12, 13], Section 3 details a formal model for the RBAC/MAC constraint checking based on typed logic and [4] to support design-time and post-design checking; Section 4 examines the Security Assurance Design Support Program and its two sub-components for design-time and post-design checking; Section 5 briefly reviews the integration of our work into the UML tool Borland's Together Control Center (TCC); and Section 6 concludes this paper. Before going into the main sections, we describe a simple example that will be used throughout the paper.

“Survey Management” Example: A Survey Institution performs and manages public surveys. After the raw data of the survey is collected, the senior staff person will add a survey header into the database. Then another staff person (senior or junior staff) will add questions into that survey, and also have the ability to categorize questions and add a new question category if needed. However, there are some special questions that have more sensitive content, with only senior staff allowed to perform data entry. Figure 1 depicts a use-case diagram for creating a new survey entry in the “Survey Management” example. The actor Staff has two children Junior Staff and Senior Staff inherited for specialization with additional extended properties. Generally, the Staff actor can perform the use-case Add Question which includes the use-case Categorize Question, and can be extended to the use-case 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 Survey Header to include a new survey header entry and the use-case Add Special Question to include special sensitive questions in a survey. Figure 2 illustrates the sequence diagram for the use-case Add Survey Header with only the main flow of events shown. To create a new survey header “Internet Usage”, the senior staff person enters data and then enables the submit button in the Survey_Header_Add_Page, which will search for the survey title in the Survey_Repository object (of class Survey_List) and then send new header data to Survey_Repository via the Add_Survey_Header message. The Survey_Repository object, in turn, creates a new survey header object Internet Usage of class Survey_Header, and then updates itself as a new item in the list of survey.

![Fig. 1. Use-Case Diagram for New Survey Entry.](image1)

![Fig. 2. Sequence Diagram for “Add Survey Header”.](image2)
denote the minimum and maximum security levels of assigned from an ordered set such as \{property, for an element \(ψ\), respectively. We consider the types (sets) \(A, UC, CI, and M\) for actor, use-case, class, and method UML elements, respectively. We denote the set of UML elements as \(Ω = A \cup UC \cup CI \cup M\) and the non-actor set \(Ω = Ω \setminus A\). In order to extend UML for incorporating the MAC property, for an element \(x \in A, UC, or M\), we denote \(x.SL\) as its security level of the element, with values assigned from an ordered set such as \([0, 1]\). For a class \(c \in CI, c.SL_{min}\) and \(c.SL_{max}\) (\(c.SL_{min} ≤ c.SL_{max}\)) denote the minimum and maximum security levels of the class \(c\), respectively.

For RBAC, we consider an actor represents one organizational role defined by the security officer, which differs from actor-use-case roles in UML [18], which are used by actors to communicate with each specific use-case. We also denote the type of UML connections as \(Ψ = \{ψ^1, ψ^2, ..., \}\) where the \(S_i\)'s refer to the names for the various UML relationships (connections) that impact security. We denote \((x; K, y; K')\) as the connection of type \(ψ^k\) from UML element \(x\) of type \(K\) to element \(y\) of type \(K'\) where \(K\) and \(K'\) are \(A, UC, CI, or M\). From [12], the \(S_i\)'s for our UML model with security are specified from (i) Use-case diagrams: (i.1). Actor inheritance: \((x; A, y; A, ψ^{A,A})\); actor \(x\) inherits actor \(y\); (i.2). Actor-use-case association: \((x; A, UC, y, ψ^{A,UC,0})\); actor \(x\) interacts with use-case \(y\) by association; (i.3). Use-case inheritance: \((x; UC, y; UC, ψ^{UC,UC})\); use-case \(x\) inherits use-case \(y\); (i.4). Use-case inclusion: \((x; UC, y; UC, ψ^{UC,UC})\); use-case \(x\) includes use-case \(y\); (i.5). Use-case extension: \((x; UC, y; UC, ψ^{UC,UC})\); use-case \(y\) extends use-case \(x\); (ii). Class diagrams: (ii.1). Class inheritance: \((x; CI, y; CI, ψ^{CI,CI})\); class \(x\) inherits class \(y\); (ii.2). Class-method defining: \((x; CI, y; M, ψ^{CM,0})\); the method \(y\) is defined in class \(x\); and (iii). Sequence diagrams: (iii.1). Use-case-class utilization: \((x; UC, y; CI, ψ^{UC,CI})\); class \(y\) is utilized in use-case \(x\) related to the concerned sequence diagram; (ii.2). Use-case-method utilization: \((x; UC, y; M, ψ^{CM,UC})\); method \(m\) is utilized in use-case \(x\) related to a sequence diagram using \(m\); (iii.3). Actor-method utilization: \((x; A, y; M, ψ^{AM,UC})\); method \(m\) is utilized by actor \(x\) in a sequence diagram; (ii.4). Method-method calling: \((x; M, y; M, ψ^{MC,MC})\); method \(x\) calls method \(y\) (via message passing in sequence diagram). We assume there is no cycle in actor and class inheritance, use-case inclusion and extension relations.

In our current model, we adopt only the Simple Security Property and Simple Integrity Property for MAC. We have MAC secure relationship constraints \((SRC)\) for relationships between non-actor elements:

- **MAC SRC** for actor-use-case association, use-case inheritance, inclusion, and extension, method-method calling and use-case-method utilization: \(SRC^M(x; K, y; K', ψ^k) = x.SL ≥ y.SL\) where \(ψ^k\) belongs to \(\{AU_{Asc}, U_{Ic}, U_{Ex}, UM_{Uz}, AM_{Uz}, and M_{Ca}\}\) for relationships mentioned above, and \(K\) and \(K'\) are \(A, UC or M\) corresponding to the types in connections: \((x; K, y; K', ψ^k)\) defined in the previous paragraph.

- **MAC SRC** for class inheritance \(SRC^C(x; CI, y; CI, ψ^{CI,CI})\) where \(x.SL_{min} ≥ y.SL_{min}\).

- **MAC SRC** for class-method defining \(SRC^C(x; CI, y; M, ψ^{CM,0})\) where \(x.SL_{min} ≥ y.SL ≥ x.SL_{max}\).

- **MAC SRC** for use-case-class utilization \(SRC^C(x; UC, y; CI, ψ^{UC,CI})\) where \(x.SL ≥ y.SL_{min}\).

The RBAC constraints are involved in working with actors: (i). Actor-use-case association: an actor \(x\) can utilize use-case \(y\) if \(x\) is allowed to use \(y\) and all other UML elements that are utilized directly or indirectly by \(y\); and (ii). Actor inheritance: an actor \(x\) can inherit actor \(y\) if \(x\) is allowed to utilize all elements that \(y\) is allowed to use.

### 3. A Checking Model for RBAC/MAC in UML

In this section, we introduce the formalization of our model for checking RBAC/MAC constraints in use-case, class, and sequence diagrams. We combine typed logic with the language \(L_{active}\) [4] which is used to model the states and behavior of active databases. Briefly, in \(L_{active}\), a fluent represents a data item with a changeable value based on time. In our work, fluents are represented as the ground instances (with values as true or false) of relations in typed logic. A fluent literal is a fluent possibly with a preceded negation symbol \(\neg\). The set of fluents that represents the current status of a system are grouped into a state (denoted as \(σ_0, σ_1, ...\) where \(σ_0\) is the initial state). A fluent \(f\) holds in a state \(σ\) iff \(f \in σ\) (otherwise, \(\neg f\) holds in a state \(σ\)). Actions (denoted with \(a\)) and events (denoted with \(e\)) both represent changes to the state of the system. We denote \(a_1; ...; a_n\) as a list of actions \(a_1; ...; a_n\) where \(a_{n+1}\) is performed after \(a_n\). For checking the validity of RBAC/MAC constraints, we define the fluents as the ground instances of the following relations:

- **Approved Link** relation \(ApL(x; x, y; ψ^k)\): The UML connection \((x; y; ψ^k) \) exists and has been approved.

- **Unchecked Link** relation \(UcL(x; x, y; ψ^k)\): The UML connection \((x; y; ψ^k) \) has been drawn but the link has not been checked (we do not know as yet whether it is approved or not).
- **Disallowable Usage** relation $\text{DisU}(x; A, y; \Omega)$: The actor $x$ has a role disallowed by customer's requirement to utilize UML element $y$.

- **Static Role-Objects Mutual Exclusion** relation $\text{ME}^{\text{RO}}(x; A, y; \Omega; z; \Omega)$: The actor $x$ has a role that is prohibited from utilizing both elements $y$ and $z$ at the same time.

- **Static Object-Roles Mutual Exclusion** relation $\text{ME}^{\text{OR}}(z; \Omega; x; A, y; A)$: The actors $x$ and $y$ have roles that are prohibited from utilizing element $z$ at the same time.

- **Link Error** relation $\text{LErr}(\cdot)$: An error flag is set due to rule violation (i.e., a constraint has been violated) on some connection.

Using the notation defined above, a design state is a set of fluents as instances of the relations $\text{ApL}(x; y)$, $\text{UcL}(x; y)$, $\text{DisU}(x; y)$, $\text{ME}^{\text{RO}}$, $\text{ME}^{\text{OR}}$ and $\text{LErr}$, which represents all of the approved links, unchecked links, allowable roles, mutual exclusions, and link errors for the UML design. Please note that in our approach, for a role-permission assignment whereas the link $x$ is prohibited from utilizing both elements $y$ and $z$ at the same time.

For example, if $\text{AddF}(x; y; \Omega)$ occurs, then the action $\text{AddF}(x; y; \Omega)$ will insert the fluent $x(A, y; \Omega, \psi_{\Omega}(\text{Add}))$ (direct assignment) or fluents $\text{AddF}(x; z; \Omega, \psi_{\Omega}(\text{Add}))$, $\text{DelF}(x; z; \Omega, \psi_{\Omega}(\text{Add}))$, $\text{DoChk}(x; y; \psi_{\Omega}(\text{Add}))$ which, in turn, triggers the constraint on the connection between UML elements $x$ and $y$. For easy reading, we add a subscript "e" next to the name of an event to distinguish from the name of an action with a subscript "a". We defined action types that affect the design states by changing their fluents: $\text{AddF}(f)$ inserts the fluent $f$ into the State table; $\text{DelF}(f)$ removes the fluent $f$ from the State table; and, $\text{DoChk}(x; y; \psi_{\Omega}(\text{Add}))$ checks the security constraint for the connection $(x; y; \psi_{\Omega}(\text{Add}))$. Then, the causal laws:

\[
\begin{align*}
\text{AddF}(f) & \text{ causes } f & \text{ if } \neg f \\
\text{DelF}(f) & \text{ causes } \neg f & \text{ if } f
\end{align*}
\]  

represent the change of state for the action $\text{AddF}(f)$ (only add the $f$ to the design state if it is not in the current state) and $\text{DelF}(f)$ (only delete the $f$ from the design state if it is in the current state). For example, if it is legal for the actor Staff to utilize the use-case $\text{Add\_Question}$ (an approved link), then the action $\text{AddF}(\text{ApL}(x; y; \Omega; \text{Add}))$ is performed to add the approved link to the design state (if it has not already been added). For reporting to the designer when an error of connection $(x; y; \psi_{\Omega}(\text{Add}))$ has occurred, we use a system action $\text{ShowErr}(x; y; \psi_{\Omega}(\text{Add}))$. Finally, we have an event definition $\text{OnChk}(x; y; \psi_{\Omega}(\text{Add}))$ caused by the action $\text{DoChk}(x; y; \psi_{\Omega}(\text{Add}))$ after $\text{DoChk}(x; y; \psi_{\Omega}(\text{Add}))$:

\[
\text{OnChk}(x; y; \psi_{\Omega}(\text{Add})) \text{ after } \text{DoChk}(x; y; \psi_{\Omega}(\text{Add}))
\]  

We can now propose the active rules that are triggered whenever a designer performs an action on a UML diagram. First, the $\text{OnDraw}(x; y; \psi_{\Omega}(\text{Add}))$ rule is used whenever a connection is made between two UML elements based on the connection type $S$;

\[
\text{OnDraw}(x; y; \psi_{\Omega}(\text{Add})) \text{ trigger } \text{DoChk}(x; y; \psi_{\Omega}(\text{Add}));
\]

\[
\text{else } \text{DelF}_{\text{Add}}(x; y; \psi_{\Omega}(\text{Add}));
\]

When the designer draws a connection (event $\text{OnDraw}$ occurs), it will trigger the action $\text{DoChk}(x; y; \psi_{\Omega}(\text{Add}))$. If the $\text{OnErr}(\cdot)$ is set, display the link error to the designer; else the system will actually draw the connection $(x; y; \psi_{\Omega}(\text{Add}))$ by $\text{Draw}$. Hence, at the interactive design time, the system tool only materializes the connection if there is no constraint violation error. If an error occurs, after the designer responds to $\text{ShowErr}(x; y; \psi_{\Omega}(\text{Add}))$ (e.g., click button “OK” in the pop-up error message), the system will abandon the attempted line for $(x, y; \psi_{\Omega}(\text{Add}))$ on the screen and remove the error $\text{Err}(\cdot)$, returning the system to the previous error-free state. In this way, the system always maintains materialized connections of the design in error-free states at design time (initially, $\text{Err}(\cdot)$ is not in the design state and it is inserted only because of the failure of $\text{Ck}(x; y; \psi_{\Omega}(\text{Add}))$ in (5) below). As we will see in Section 4, the action $\text{DoChk}(x; y; \psi_{\Omega}(\text{Add}))$ can be invoked directly by the designer on unchecked links or by the post-design checking program. Further, by (3), whenever $\text{DoChk}(x; y; \psi_{\Omega}(\text{Add}))$ is invoked, it generates an event $\text{OnErr}(x; y; \psi_{\Omega}(\text{Add}))$ which, in turn, triggers other (conditional) actions as listed below:

\[
\text{OnErr}(x; y; \psi_{\Omega}(\text{Add})) \text{ trigger if } \text{Ck}(x; y; \psi_{\Omega}(\text{Add})) \text{ then }
\]

\[
\{ \text{AddF}_{\text{Add}}(\text{Ck}(x; y; \psi_{\Omega}(\text{Add}))); \text{AddF}_{\text{Add}}(\text{ApL}(x; y; \psi_{\Omega}(\text{Add}))) \}
\]

\[\text{else } \text{AddF}_{\text{Add}}(\text{Err}(\cdot))\]

(5)

where $\text{Ck}(x; y; \psi_{\Omega}(\text{Add}))$ (elaborated below) is the Check Link constraint to verify whether the connection $(x; y; \psi_{\Omega}(\text{Add}))$ is legal. If $\text{Ck}$ is true, then we delete the fluent for the unchecked link, and add the fluent for the approved link. Otherwise, a link error fluent is set.

Second, the active rule $\text{OnUncheck}$ is used whenever the designer removes the connection $(x; y; \psi_{\Omega}(\text{Add}))$ of type $S$, that exists between two UML elements $x$ and $y$:

\[
\text{OnUncheck}(x; y; \psi_{\Omega}(\text{Add})) \text{ trigger } \text{DelF}_{\text{Add}}(\text{Ck}(x; y; \psi_{\Omega}(\text{Add}))); \text{DelF}_{\text{Add}}(\text{ApL}(x; y; \psi_{\Omega}(\text{Add})))
\]

(6)

which triggers the action of deleting the fluent for the unchecked link (if it has not been checked) and deleting the fluent for the approved link (which represents the connection and no error has occurred). Note from (4), (5) and (6), if the design is created from scratch, the
system always maintains the design in an error-free state. Hence, when the designer erases a connection as in (6), the OnErase event does not affect the error-free property of the current design state; thus we do not need to be concerned with incremental deletion effect by reconsidering other connections.

In order to precisely specify CkL, we define two intermediate relations. The first one is Legal Use:

\[ LgU(x; \Sigma, y; \Omega) = [3 \forall \psi] \left( \text{ApL}(x; \Sigma, y; \psi^d) \lor [3 \forall \psi, \exists : \Sigma ApL(x; \Sigma, z; \psi^d), LgU(z; \Sigma, y; \Sigma)] \right) \]  

which states that x can use y if x has an approved link to y or to some z which can use y. For example, in UML, if actor x is approved to utilize use-case z and z validly includes use-case y, then x can use z legally. Legal Use tracks both the direct connections (approved links) and the indirect connections (resulting from other UML connections between elements). The second intermediate relation is Prohibited:

\[ Prh(x; A, y; \Omega) = \text{DisU}(x; A, y; \Omega) \lor [3 \forall \psi] \left( \text{ME}^{SRO}(x; A, \Omega, z; \psi) \lor LgU(x; A, z; \Omega) \lor [3 \forall \psi, \exists : A \text{ME}^{SRO}(\Omega, x; A, z; A), LgU(z; A, y; \Omega)] \right) \]  

which states that the actor x is prohibited from using UML element y. This can occur due to one of the three situations: (i) its role has been directly prohibited by the customer’s security requirement, (ii) x is not allowed to use both y and some element z at the same time due to the Role-Objects mutual exclusion ME^{SRO}(x; A, y; \Omega, z; \psi), or (iii) x conflicts with some actor z that can use y by LgU(y, x) but has an Object-Roles mutual exclusion relation with x on y by ME^{SRO}(y; \Omega, x; A, z; A).

Given the prior definitions, the Check Link constraint (element x connected to element y based on connection type S) is defined as the conjoin of CkL^{M} (for MAC) and CkL^{R} (for RBAC):

\[ CkL(x; \Sigma, y; \Sigma, y; \psi^d) = CkL^{M}(x; \Sigma, y; \Sigma, y; \psi^d) \land CkL^{R}(x; \Sigma, y; \Sigma, y; \psi^d) \]  

From Section 2, the MAC constraint checking CkL^{M} verifies the domination of the security level of x over the security level of y:

\[ CkL^{M}(x; \Sigma, y; \Sigma, y; \psi^d) = SRC^{M}(x; \Sigma, y; \Sigma, y; \psi^d) \]  

where SRC^{M}(x; \Sigma, y; \Sigma, y; \psi^d) is defined in Section 2. For example, CkL^{R}(Staff; A, Add_Survey_Header; UC, \psi^{UC,Asc}) fails since the security level of Staff (Confidential-C) does not dominate the level of use-case Add_Survey_Header (Secret-S) (see Figure 1). The definition of CkL^{R} varies by each connection type \psi^d. For direct AU,Asc connection, and implied actor to a non-actor typed-K (K is UC, Cl or M) element (that is, since a use-case can be connected to use-cases, classes, methods, etc., an actor can also be connected to them, if the actor is connected to a use-case):

\[ CkL^{R}(x; A, y; K, \psi^{K, y}) = [-Prh(x; A, y; K)] \land (\forall \Omega) \left( LgU(y; \Omega, K, z; \Omega) \Rightarrow -Prh(x; A, z; \Omega) \right) \]  

This means that actor x can legally utilize a non-actor element y of type K if and only if x is not prohibited to do so and also not prohibited to use any element z that y can legally use. For example, in Figure 2, suppose we have approved that method OnSubmit can call methods Survey_Title_Search and Add_Survey_Header which, in turn, calls Create_Survey_Header. Then, in order for actor Senior_Staff to be allowed to use method OnSubmit, we need to check whether there is any constraint fluent that prohibits Senior_Staff to use the methods Survey_Title_Search, Add_Survey_Header and Create_Survey_Header. For actor inheritance:

\[ CkL^{R}(x; A, y; A, \psi^{A,y}) = (\forall \Omega) \left( \text{ApL}(y; A, z; \Omega) \Rightarrow CkL^{R}(x; A, z; \Omega, \psi^{A,y}) \right) \]  

This means that actor x can be a legal child of actor y if and only if for any element z that has an approved link from y, there is nothing to prohibit x to use z legally. For example, suppose the parent actor Staff is approved to use use-cases Add_Question, Categorize_Question, and Add_Question_Category (see Figure 1), then when we create the actor Junior_Staff, in order for Junior_Staff to be a child of Staff, we must check if Junior_Staff is not prohibited from Add_Question, Categorize_Question, and Add_Question_Category. Finally, for use-case inheritance, use-case inclusion, use-case extension, use-case-class utilization, use-case-method utilization, class-method defining, and method-method calling:

\[ CkL^{R}(x; \Omega, y; \Omega, \psi^{\Omega}) = (\forall \Omega) \left( LgU(y; \Omega, \Omega, z; \Omega) \Rightarrow CkL^{R}(x; \Omega, z; \Omega, \psi^{\Omega}) \right) \]  

This means that for the connection from x to y to be approved, we must check that any actor z that has been allowed to use x should also be allowed to use y and any other element \psi legaly used by y.

4. Security Assurance Design Support Program

The intent of the formal framework, as presented in Section 3, is to support two modes of RBAC/MAC constraint checking: design-time for whenever the designer alters a UML diagram by adding/deleting connections and post-design for design milestones or versions in an application’s UML diagrams for checking across all application (diagram) connections at once. Design-time and post-design constraint checking are realized, algorithmically, as two parts of a Security Assurance Design Support (SADS) program component. The SADS program component has been designed to be integrated into a UML tool, which for our purposes will be Borland’s Together Control Center (TCC), as we will discuss in Section 5. The SADS has two sub-programs: dSADS for the design-time mode and pSADS for the post-design mode. These two sub-programs share different rules (1) to (6)) and constraints (7) to (13) of Section 3, while they differ in their utilization of the rules and constraints, i.e., dSADS utilizes specific ones when a connection is added or deleted in a UML diagram, while pSADS tends to utilize all of them programmatically across the entire UML design.
The job of dSADS is similar to that of the Execution Monitoring Security Automata (EMSA) by [23] and Edit Automata (EA) by [21]. The main difference is that EMSA and EA are used for monitoring the execution steps of an application to enforce the security policies whereas dSADS monitors the stage of the design process of an application to enforce the design outcome satisfying the customer’s security requirements.

The design-time security assurance design support program is intended to be always active within the design environment (e.g., UML tool) whenever a designer is creating/choosing a UML design.

Definition 4.1. A Design-time Security Assurance Design Support (dSADS) program $P^{DT}$ is a program to perform rules (1) to (6) and a collection of constraints (7) to (13) with an initial state of design input.

In order to track security concerns for each design state, we introduce four design state categories:

Definition 4.2. A design state is said to be: dubious if it contains an Unchecked Link fluent $uCL$; violating if it contains a Link Error fluent $LErr$; safe if it is neither dubious nor violating (otherwise called unsafe); and live if it has some approved links (Approved Link fluents $ApL$)\(^1\).

The categories are intended to provide information to the designer on the design state with respect to security, alerting the designer to the status as changes are made to the UML diagrams.

To illustrate the design states and the $P^{DT}$ program, we consider a typical scenario of usage. Suppose that a designer working with UML for a new application starts with an empty design state (there are no fluents). As the designer creates a design (adding new UML elements, and most importantly for our purposes, connecting them to one another), the series of actions that are performed enable $P^{DT}$ to actively generate corresponding events and actions (as described in Section 3) depending on the current state (thereby adding and deleting fluents). These events may have no impact (adding an actor without connecting it to a use-case or another actor via inheritance), may change it to a new state without link errors (when adding a connection), or may generate a link error without materializing the connection in the design. In the last case, the designer may resolve the problem by finding and removing the source of the conflict (e.g., due to a previous connection), by changing the security level ($CkLM$ in $CkL$ may have been violated as a result of the connection) and then re-drawing that link, or simply by abandoning the intention to connect that link. For the designer starting from scratch, our objective in our implementation of $P^{DT}$ in a UML tool is to not allow connections that cause link errors related to security if the design-time checking is enabled. Throughout the process, the designer’s goal is to always try to maintain a safe (no link errors) and live (approved links between UML elements) design state regardless of the number of actions that s/he has taken. $P^{DT}$ is intended to ensure that the state is always safe.

Clearly, the advantage of $P^{DT}$ coupled with its realization with a UML tool like TCC provides a robust and interactive working environment for the designer. At each step, the designer is informed of the link error immediately as it occurs (with the connection prevented if desired). Such instant alerts allow the designer to correct the problem (e.g., change a security level) to maintain a safe and live state. However, in some cases the constraint check within $P^{DT}$ may not be trivial and simply involve the connection of two UML elements, i.e., each may be connected with many other elements. In situations such as rule (13), the time for the check may be unacceptably long, as recursive links among design elements are checked. Further, our scenario has assumed that a designer has started from scratch and incrementally constructed his/her design, always insuring that link errors are corrected as they occur (connections are prohibited if $P^{DT}$ is enabled in a running environment). However, in practice, the scenario can be more complex; a designer may start with a safe state and import one or more unsafe diagrams into his/her design, or may turn off $P^{DT}$ if the design connections are too complex to always maintain a safe state from a security perspective. In these situations, it will be necessary to employ the post-design program $P^{PD}$. In this case, the designer incrementally changes his/her UML design without checking the connections as they are added and works until a meaningful milestone or version is reached. The version/milestone to be checked can have a combination of checked and unchecked links, e.g., a version which was previously checked to which new links have been added, a version that unifies two or more versions, some of which may be checked, etc. It is against this version that the post-design checking program $pSADS$ can be executed to perform the overall constraints checking, which requires the checking of unchecked links, and the re-checking of checked links which may now be invalid due to new unchecked connections.

The post-design security assurance design support program is intended to be explicitly invoked by a designer within the design environment (e.g., UML tool) at regular version/milestone intervals. The program, as given below, arranges the order of the checking of links to minimize recursion:

Definition 4.3. A Post-Design Security Assurance Design Support ($pSADS$) program $P^{PD}$ is a program that performs the checking of links with a dubious UML design state as input (including all role constraints $AR$ and $ME$) in the following steps:

1. Invoke the $\text{DoChk}_{\{x, y : \Omega, \psi\}}(\Omega, y : \Omega, \psi)$ on all of the connections between non-actor elements ($x : \Omega, y : \Omega, \psi$).
In this section, we briefly report on our prototyping effort on transitioning RBAC, MAC, and the Security Assurance Design Support (SADS) program and its two sub-programs dSADS and pSADS, as described in Section 4, into Borland’s UML tool Together Control Center (TCC). TCC provides Open APIs and a plug-in structure, which has allowed us to incorporate our security properties and algorithms as presented in Sections 2, 3 and 4 into UML across the different diagrams, and to include custom Java code that realizes the rules (1) to (6) and constraints (7) to (13) (Section 3), to dynamically analyze the security as a UML design is constructed, supporting design-time constraint checking. To illustrate, in Figure 3, actor $a_1$’s security property contains role $Senior$ stuff with $SL$ (called $CLR$) = $S$ (Secret), where the UML’s $Security$ property display has been altered with our own custom code. Consider Figure 4 and assume that $dSADS$ is enabled; if we attempt to connect actor $a_2$; $Staff$ to use-case $uc_2$; $Add$ Survey Header, an error dialog is displayed, since the $Staff;SL = C < uc_2; SL = S$ violates the MAC constraint (10). The security plug-in also inserts comments of security levels into the generated code, allowing security to be tracked into later development and maintenance stages. In summary, at this time we have partially implemented $dSADS$ and are in progress on implementing $pSADS$.

5. Prototyping Effort

In comparing the work in Sections 3 and 4 with other efforts, we believe it is necessary to make a number of observations. To make it easier to explain the concepts, assume that we have two actors $ac_1$ and $ac_2$ that are mutually exclusive on use-case $uc$, and that we first connect $ac_2$ to $uc$. When we then try to connect $ac_2$ to $uc$ we will have a link error on the connection attempt. Given this scenario, let us consider the approaches of [7, 16] and the way that they each handle this situation as compared to our work. In [16], they support only 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 – we’ll assume that the $ac_1$ to $uc$ connection is chosen by rule. In [7], while they only support post-design, they provide all of the relevant alternatives when conflicts are identified, which means that the designer can either choose $ac_1$ to $uc$ or $ac_2$ to $uc$. In comparison to our own work, our post-design conflict checking aligns closely to the work of [7]; we report the ME conflict which can then be resolved per the designer’s preference. Further, our design-time checking is more robust in the sense that the conflict on the second attempted connection ($ac_2$ to $uc$) alerts the error.

6. Conclusions and Ongoing Research

This paper has proposed a formal model for design-time and post-design constraint checking of unified modeling language (UML) diagrams which have been extended with RBAC/MAC properties and concepts. To accomplish this: Section 2 reviewed background on MAC, UML, typed logic and active database language $L_{active}$; Section 3 introduced a checking model for RBAC/MAC constraints on UML design represented in $L_{active}$; and, Section 4 detailed the integration of the work of Section 3 for design-time and post-design
security assurance, which has been transitioned as a prototype into Together Control Center as reviewed in Section 5. We believe that the approach and formal model presented in this paper will help the designer to attain a safely secured design (as a set of UML diagrams augmented with consistent RBAC/MAC constraints) in support of Stages 1 and 2 as outlined in Section 1, which can then serve as a prelude to validation in Stage 3, which is the subject of our ongoing work.

One critical issue that is in progress is to theoretically demonstrate that safety is achieved, in terms of an absence of link errors, for UML designs that are created from scratch using our approach. This is the subject of ongoing work where we wish to prove that the MAC properties (simple security and simple integrity) are never violated if equation (10) holds. The safety of RBAC is relative with respect to the constraint facts provided by the customer (realized in our work via Disallowable Usage and Mutual Exclusion fluents). If the customer neglects to provide a constraint fact (e.g. a missing mutual exclusion condition in the requirement), the design outcome is not practically safe. Therefore, we can only formally prove the RBAC satisfaction in terms of "no link errors", then move to Stage 3 to verify the completeness of the security requirement with the terms of an absence of link errors, for UML designs that attain a safely secured design (as a set of UML diagrams).

To further support the capabilities of dSADS and pSADS (Section 4) by using constraint logic programming tool to verify the security design states.

7. References


