15

Software architectural alternatives for user role-based security policies

S. A. Demurjian, Sr., T. C. Ting, and J. A. Reisner
Computer Science & Engrg. Dept., The University of Connecticut
191 Auditorium Road, Storrs, Connecticut 06269-3155, USA,
860.486.3719, 860.486.4817, {steve,ting,reisner}@eng2.uconn.edu

Abstract

Security concerned users and organizations must be provided with the means to protect and control access to object-oriented software, especially with an exploding interest in designing/developing object-oriented software in Java, C++, and Ada95. Our user-role based security (URBS) approach has emphasized: a customizable public interface that appears differently at different times for specific users; security policy specification via a role hierarchy to organize and assign privileges based on responsibilities; and, extensible/reusable URBS enforcement mechanisms. This paper expands our previous work in URBS for an object-oriented framework by exploring software architectural alternatives for realizing enforcement, with the support of assurance and consistency as a key concern for security policies that evolve and change.

Keywords

Software architectures, object-oriented, enforcement mechanisms

1 INTRODUCTION

How will assurance and consistency be attained during the definition and usage of an application's user-role based security policy, particularly in an object-oriented context that stresses change and evolution? This question is interesting, particularly with the explosive growth of object-oriented software development. While C++ has been a strong player since the late 1980s, Ada95 and Java offer new opportunities that are targeted for diverse and significant market segments. Security has and will continue to be a major concern, especially in Java, where security must be present to control the effects of platform-independent software. Health care systems require both high levels of consistency and assurance, while simultaneously needing instant access to data in life-critical situations. In CAD applications, the most up-to-date specifications on mechanical parts must be available in a shared manner to

Database Security XI T.Y. Lin and Shelly Qian (Eds)

promote cooperation and facilitate productivity, making consistency and assurance important from a business perspective.

Over the past few years, we have concentrated on discretionary access control, by defining a user-role based security (URBS) model that can be utilized in the design and development of object-oriented applications. The current public interface provided by most object-oriented languages is the union of all privileges (methods) needed by all users of each class. This allows methods intended for only specific users to be available to all users. For example, in a health care application (HCA), a method placed in the public interface to allow a Physician (via a GUI tool) to prescribe medication on a patient can't be explicitly hidden from a Nurse using the same GUI tool. Rather, the software engineer is responsible for insuring that such access does not occur, since the object-oriented programming language cannot inherently enforce the required security access. We have proposed a user-role definition hierarchy (URDH) to organize responsibilities and to establish privileges. Privileges can be assigned (can invoke a set of methods) or prohibited (cannot invoke a set of methods) to roles, thereby customizing the public interfaces of classes on a role-by-role basis. Our recent efforts have proposed extensible and reusable URBS enforcement mechanisms, with the goal to minimize the amount of knowledge a software engineer must have on URBS. Work on the object-oriented design model [5] and URBS [1, 4] have been published.

This paper expands our previous work to include assurance and consistency, particularly since we are committed to a continued exploration of automatically generated URBS enforcement mechanisms. Since class libraries may not offer a secure enough venue to insure high consistency and assurance, we have turned to the field of software architectures to investigate potential solutions to augment our previous enforcement mechanisms work [2, 3]. Software architectures [6] expand traditional software engineering by looking at how the major components of a system can mesh and interact. This is especially relevant for object-oriented software, where a class library for a problem is initially developed, with software engineers designing and building tools against that class library to implement the overall capabilities of an application. In such a model, the URBS enforcement mechanism must interact with both the class library and the tools for an application, to insure that users utilizing tools only access those portions of the application on which they have been granted access. In our approach, this translates to the users only being able to invoke methods that have been authorized to their respective roles.

The remainder of this paper contains three sections. In Section 2, we discuss the critical need of consistency for security, as we seek to guarantee a level of assurance to designers and users utilizing an URBS/object-oriented approach. In Section 3, we propose and explore software architectural variants that can offer varying degrees of assurance and consistency, and critique the variants by comparing and contrasting their capabilities from multiple perspectives. Section 4 concludes this paper.

2 THE NEED FOR CONSISTENCY AND ASSURANCE

Role-based security policies and enforcement mechanisms must have high consistency in order to support a high assurance, secure system. The consistency must be maintained at all levels within the policy, including individual roles, role hierarchies, and end-user authorizations, to insure that their creation, modification, and deletion will always maintain the required URBS policy. Consistency is the foundation upon which high integrity and assured secure systems must be built. A set of techniques/tools must be provided that allow URBS policies to be analyzed and assured at all times during design, development, and maintenance of object-oriented software.

In general, URBS policies are application dependent, and consequently, data security requirements vary widely from application to application. For example, sensitive health care data must be both protected from unauthorized use while simultaneously be almost instantaneously available in emergency and life critical situations. On the other hand, in some design environments such as CAD, the most up-to-date specifications on mechanical parts must be available in a shared manner to promote cooperation. In this case, the URBS policies may not protect sensitive personal information, but may protect information which is equally sensitive from a business perspective.

The ultimate responsibility for URBS policies is on the shoulders of the application's management personnel and organization's data security officer. In order to have these critical policy makers take full advantage of URBS, tools and techniques must be made available. Design techniques are critical to allow software and security engineers to accurately and precisely specify their applications' functional and security requirements. To augment these techniques, a suite of tools is required, that can provide many different and diverse analytical capabilities. These tools should automatically alert these engineers when potential conflicts occur during the creation or modification of roles, role hierarchies, and end-user authorizations, thereby heading off possible inconsistencies. There must also be tools that provide on-demand analyses, allowing engineers to gauge their realized software and/or security requirements against their specifications. Once the URBS policy has stabilized, the tools should provide the means to capture and realize it via a URBS enforcement mechanism that is automatically generated. The overriding intent is to finish with an object-oriented system that embodies a strong confidence with respect to the URBS policy and its attainment.

The remainder of this section explores these and other issues from two perspectives. In Section 2.1, we examine the consistency issues that must be attainable as roles and dependencies among roles are created and modified. In Section 2.2, we investigate similar consistency issues as actual individuals (people) are authorized to play certain roles within an object-oriented application or system.

2.1 Consistency for User Roles

When a security engineer is creating and modifying user roles for an object-oriented application or system, the consistency of the definition is critical in order to insure that the URBS policy is maintained. This is a time-oriented issue; changes to the policy are needed, especially in object-oriented situations, where evolution and extensibility are the norm. Regardless of the changes that are made, there must be assurance that the privileges of each user role are adequate to satisfy the functions of the user role. Moreover, the privileges must not exceed the required capabilities of the user role, to insure that misuse and corruption do not occur. In addition, since user roles are often interdependent upon one another (e.g., our approach uses a hierarchy), it may be necessary to examine their interactions to insure that privileges aren't being passed inadvertently from role to role, yielding a potentially inconsistent state.

There are many different scenarios of evolution that must be handled. A security engineer may create new roles for a group of potential users or may create specific roles that are targeted for a particular end-user for a special assignment under a special circumstance. Each newly created role must be *internally consistent* so that no conflicts occur within the role itself. This is also true when a role is modified, which we term *intra-role consistency.* For the object-oriented case, when privileges are assigned to each role, this assignment implicitly grants object-access privileges to the role holder (end-user). Such an assignment process utilizes the *least-privilege principle* which grants only necessary access privileges but no more. Only those privileges that are relevant to the user role are permitted. The policy is intentionally very conservative and restrictive, requiring that the URBS policy be validated by either the software engineer, security engineer, or both. In some organizations, there are dedicated security officers who possess the ultimate responsibility with respect to security requirements/policies for all applications.

To complement the least-privilege principle, user roles often must satisfy *mutual exclusion conditions.* Here, there must be a careful balance between permitting access to certain objects while simultaneously prohibiting access to other, special objects. For example, in HCA, an individual assigned the role of `Pharmacist` can read the prescription of a patient, update the number of refills after processing the prescription, but is explicitly prohibited from modifying the dosage or drug of the prescription. Thus, access and modification to some information is balanced against exclusion from other information. This strong mutual exclusion situation is clearly observed by the medical profession and is mandated by law. The URBS policy must ensure that security requirements such as these are not violated. In our approach, these mutual exclusions are supported in the URDH by allowing the security engineer to define prohibited methods, which provides the means to insure that the prohibited privileges of a role do not contradict with the assigned privileges.

When one extrapolates to consider the interdependence of user roles, such as

within our URDH, the internal consistency as captured by least privilege and mutual exclusion, must be expanded to ***inter-role consistency.*** In any approach with interdependence among user roles, there is the potential for user roles to acquire privileges (both positive and negative privileges) from other roles. In addition, to provide versatile design tools to the security engineer, it should be possible to establish superior, inferior, and equivalence relationships among different user roles. These relationships must also be validated as privileges are defined, acquired, and change. From the perspective of the entire URDH, *intra-hierarchy consistency* must be attained.

To support a URBS definition process with least privilege and mutual exclusion, the security engineer must be provided with a set of techniques and tools. There must be tools for meaningful comprehension on user roles, including all positive privileges, negative privileges, and relationships to other roles, supporting intra-role consistency. Once any initial definition has occurred, there must be tools to support analyses for both internal and inter-role consistency. Automated analysis tools are necessary for an exhaustive search to follow all possible object access paths as required by all of the positive and negative privileges in the security definition. Conflicts discovered during the search will have to be resolved by the application's management personnel and security engineer. Feedback must be available to assist the human designer in arriving at a viable resolution to any conflicts or inconsistencies. Analyses are available in the ADAM environment for the application's content/context, and for its security requirements [4, 5]. Capabilities analyses allows one to review the permissions given to a chosen URDH node on an application's OTs, methods, and/or private data, supporting the intra-role or internal consistency of the URBS policy. Authorization analyses allows one to investigate which user roles have what kinds of access to an OT, a method, or a private data item, supporting inter-role and intra-hierarchy consistency.

2.2 Consistency in End-User Authorization

When considering consistency in end-user authorization, the assumptions of the policy must be clearly understood. For example, in any organization where end-users can be assigned multiple roles, there are two scenarios of permissible behavior against an application: 1. end-users can only play exactly one role at any given time; and, 2. end-users can play multiple roles concurrently at any given time. The first assumption does not cause significant problems, since for an end-user, only one role is active. As long as that role is intra-role and inter-role consistent, there is no problem. However, the first assumption alone does not provide the needed security, but instead raises a number of interesting issues that are addressed by the second assumption.

Namely, when an end-user may play multiple user roles simultaneously at any given time within an application and within the organization, a level of *end-user consistency* in introduced. In end-user consistency, the privileges of the multiple roles for are aggregated, which may introduce conflicts between

positive and negative privileges that span multiple roles. Further, when a new privilege is assigned to an established user role, with internal and inter-role consistency assured, it may still impact the end-user consistency. Automated tools are needed for the user authorization model in a secure data system so that no URBS policy violations are possible for any end-user in the organization. Thus, the techniques/tools in Section 2.1 must be extended to consider end-user consistency, allowing the security engineer to focus on the conflicts of privileges for single end-users with multiple concurrent roles.

Once intra-role, inter-role, and intra-hierarchy, and end-user consistency have been attained at a definitional level, there are two remaining requirements: (1) the defined URBS policy must be captured within the object-oriented application; and (2) once captured, at both compile time and run-time, the policy must be enforced. For both requirements, our previous work on URBS enforcement approaches [2, 3] is intended to support, in part, the consistency and assurance of the URBS policy. However, as we will see in Section 3, through software architectures we can provide a higher level of assurance regarding the guarantee that must be met concerning a defined URBS policy for an object-oriented application.

3 SOFTWARE ARCHITECTURES AND URBS MECHANISMS

To understand our efforts in this section, it is critical that we define our assumptions concerning the composition of object-oriented software. Basically, the crux of an object-oriented system is an underlying, shared object type/class library to represent the kernel or core functionality. Once such a library has been developed, other software engineers will use it to design and develop tools. Thus, end-users are not able to write programs to access data directly. Rather, end-users utilize tools that embody the apropos security code to enforce the required URBS policy.

Software architectures [6] is an emerging discipline whose intent is to force software engineers to view software as a collection of interacting components. Interactions occur both locally (within each component) and globally (between components). In understanding interactions, the key consideration is to identify the communication and synchronization requirements which will allow the functionality of the system to be precisely captured. By taking a broader view of the problem definition process, software architectures permits database needs, performance/scaling issues, and security requirements, to be considered. These considerations are critical as large-scale object-oriented software design, development, and usage becomes increasingly dominant.

Our purpose in this section is to present and critique multiple software architectural variants for URBS enforcement of object-oriented software, with a constant focus on the attainment of consistency and assurance. Our intent in this section is to step back from our previous work [3] and consider the ways that these approaches can fit into an overall architectural scheme to

security enforcement for object-oriented software. Nevertheless, we start this section by briefly reviewing two of our previous approaches, since they set the context for our subsequent discussion related to software architectures. Then, we focus on two architectural styles: layered systems, which are most known from ISO layers; and, communicating processes, which underlie today's client/server paradigm. For both styles, multiple variants are presented and analyzed. The final section critiques the six different variants by comparing and contrasting their capabilities.

3.1 UCLA and GEA Approaches

The *URDH-class-library approach (UCLA)* employs inheritance to implement the enforcement mechanism by creating a class hierarchy for the URDH. For each URDH node, positive method access is based on the defined assigned methods. At runtime, a user's role guides the invocation of the appropriate methods that are used to verify whether the user's role has the required permissions. From an evolvability perspective, as user roles are added, or as privileges are changed, only the URDH class library must be recompiled.

The *generic exception approach (GEA)* utilizes reusable template classes to realize a significant core of generic code that encapsulates the URBS policy. In GEA, when a method is invoked, the user role of the current user is checked to verify if access can be granted. If the user's role doesn't permit access, an exception is thrown, and the invoking method will not allow its functionality to be executed and affect instances. The code in the GEA security template is hidden from the software engineer. Software reuse is promoted since the template is reused by all classes that require URBS enforcement.

3.2 Architectural Alternatives

From a software architecture perspective, the URBS enforcement mechanism can be located in many different places and function in many different ways. It can be integral part of the OT/class library, to be automatically included when any tool utilizes a portion of the library. Alternatively, it may be an independent and self-contained library that is compiled with each application tool, similar in concept to a math library being included. Other choices could have a separately executing process through which all security requests must be handled. Regardless of the choice, the key underlying characteristics must be the attainment of high consistency and assurance. This must be balanced against the need to minimize the amount of knowledge a software engineer must have on URBS and to have an approach that is evolvable, since object-oriented software and its security policy will change over time.

To standardize terminology regarding the assumptions on object-oriented systems given in the introduction of Section 3, we define:

AppCL: Represents the shared, object-oriented class library for an application.
SCL: Represents the security class library for an application that embodies URBS definition and enforcement.
TCL: Represents the tool class library for individual tools (e.g., a patient GUI, an admissions subsystem, etc.) against the application.

Note that when the L is dropped from either AppCL or SCL, we are referring to an individual class of the library. The remainder of this section explores layered systems, and communicating processes and the client/server paradigm, as software architectural alternatives for URBS enforcement. For each alternative, multiple variants are presented and discussed, and then analyzed with respect to: the level of consistency and assurance that each variant provides for security concerned users; the dimensions of evolvability, which is critical since both the URBS policy and object-oriented software tend to be dynamic over time; and, the impact of the absence/presence of a persistent store.

(a) Layered Systems

Layered systems are a classic technique for software architectures, where layers of functionality are built upon one another to provide a controlled environment for access to information. In Figure 1, there are two layered system variants for URBS enforcement: LS1 an application-based approach on the left, and LS2 a class-based approach on the right. In both variants, security is at the level of the method invocation, which is processed by the SCL prior to its actual runtime call against an instance of a class in the AppCL. In either case, the SCL can be either the UCLA or GEA approach. In the LS2 variant, each individual class handles the method invocations that apply to its instances as they are received by the various tools. The difference is one of granularity. In LS1, security is managed at the application level overall, and once it has been determined that the tool can invoke a method, it is passed through to the involved instance or instances. In LS2, security is managed at the instance level only. This may cause a problem when instances refer to other instances, i.e., a security request by the tool involves multiple instances of either the same or different classes.

From a consistency/assurance perspective, it appears that LS1 has the advantage, since all of the method invocations must pass forward through the security layer for authentication and all results must pass back for enforcement. That is, when utilizing a tool and its various options, users end up calling various methods based on his/her UR and under the control of the tool. However, variant LS2's view of allowing each instance to maintain its own security is superior to LS1 from a software evolution perspective, since changes to the security policy of one class may not effect the policies of other

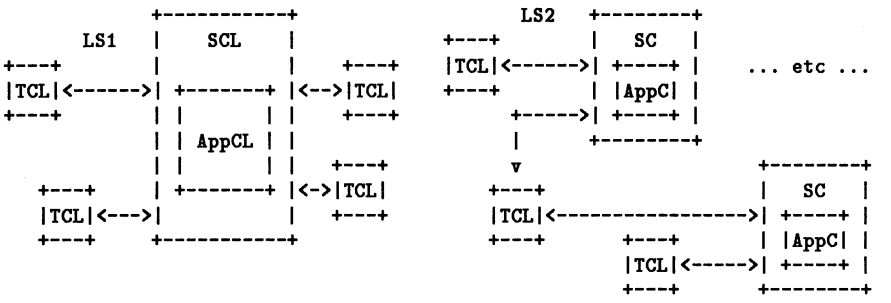

Figure 1 Application-Based (LS1) and Class-Based (LS2) Approaches.

classes. When a persistent store is included, LS1 has the edge, since all accesses must proceed via a common security layer. In LS2, there are potential concurrent access issues if some or all AppCs are connected to a database.

(b) Communicating Processes - C/S Paradigm

In the *communicating processes* approach to URBS enforcement, a process-oriented, client/server (C/S) paradigm is adopted. TCL, SCL, and AppCL are integrated into single and/or multiple processes, resulting in a total of four different variants: CP1, CP2, CP3, and CP4. The variants differ in their number of processes and the grouping of the TCL, SCL, and/or AppCL into various processes. Note that for both CP1 and CP2, each SCL and/or AppCL represents the minimal subset needed by the tool/TCL to support its functionality and enforce its security policy.

In Figure 2, variant CP1 is given, which is similar to LS1, except that each tool is compiled as a separate, standalone process. In this case, SCL and AppCL are analogous to a math library that is compiled when needed by the software. Functionally, within each process, TCL sends method invocations to SCL which in turn passes them through to AppCL according to the URBS policy. Results are passed back from AppCL to SCL, which may then filter the response before passing them back to TCL. Note that SCL and AppCL in each process represents those subsets of the class libraries needed by each tool, and that either UCLA or BEA can be the enforcement mechanism realized within SCL.

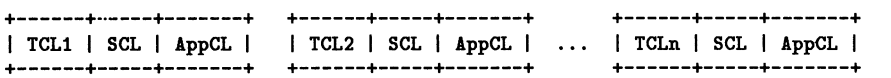

Figure 2 CP1: A Single Process, Non-C/S Approach.

From a consistency/assurance perspective, it would be a requirement that each tool be compiled into a single process with SCL and AppCL included. Thus, the level of assurance and consistency that is attained is tied to the accuracy

and completeness of the URBS policy. But, note that since each tool may have a only a portion of the overall URBS policy, consistency becomes a prominent concern whenever changes need to be made, i.e., updates must be made to all tools that use the portion of the policy that changed. Extensibility in CP1 presents major problems. While it is easy to add new tools, and new tools when added won't effect existing tools, changes to either the SCL or AppCL definitely cause problems. If changes to the SCL are localizable to data files that can be dynamically loaded, then URBS policy changes should be supportable. But, if the changes require the SCL to be rebuilt, unless the compilation/runtime environment supports dynamically linkable class libraries, all affected tools must be recompiled. Changes to AppCL have a dramatic impact for all affected tools and all affected portions of the SCL. In addition, since the AppCL is compiled with each tool, it is unclear whether this approach can successful work when AppCL is linked to a database.

Variants CP2 and CP3, shown in Figure 3, are both multi-process approaches with clearly defined client/server separation of functionality. In CP2, shown in the left side of Figure 3, each client is a TCL/SCL pair that interacts with a shared AppCL server. In this case, each SCL represents that subset of the overall URBS policy/enforcement that is needed by the specific tool, i.e., if a tool only uses one or two classes, the SCL is that subset of the overall URBS policy for those needed classes. Thus, the URBS policy/enforcement is specifically bound to each tool. Like CP1, the level of consistency/assurance that is attained depends on the realization of the URBS policy within SCL. The fact that the policy is spread across multiple tools does introduce potential consistency concerns when changes to the policy are made. Changes to the URBS policy impact SCL in the same way as CP1. However, there are improvements in changes to AppCL; since it is in a separate process, careful planning will allow some changes to have no impact on the joint TCL/SCL clients. Drastic changes to AppCL (e.g., deletion of classes, additions of classes, major functionality upgrades) are likely to impact SCL thereby requiring the recompilation of tools. UCLA and GEA are tightly linked to AppCL, making them inappropriate for CP2. From a database perspective, the presence of a persistent store within or coupled to AppCL should be supportable and invisible to the clients.

In CP3, shown in the right side of Figure 3, the client is each individual tool (TCL), with the server containing the joint SCL/AppCL functionality. By decoupling the URBS policy/enforcement from each tool, the tool becomes relatively independent from changes to the security policy. Each tool simply makes requests to the joint server and the way that those requests are satisfied can be hidden using typical object-oriented design approaches. Thus, unlike CP1 and CP2, changes to the URBS policy shouldn't impact tool code. The placement of the entire URBS policy/enforcement in one location greatly improves consistency and assurance, since all changes to the policy occur in one place. This is superior to both the CP1 and CP2 variants. Like CP1, SCL can be realized with UCLA or GEA.

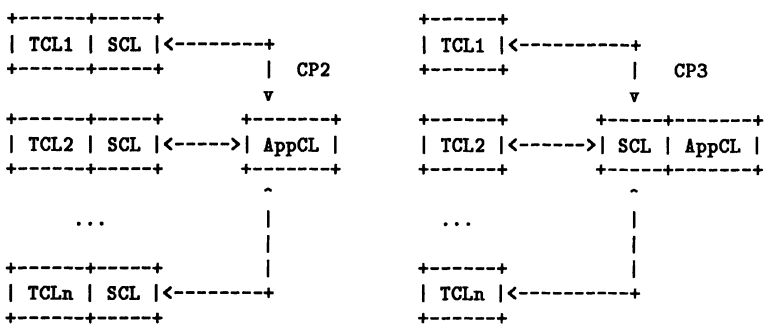

Figure 3 Multi-Process C/S Approaches.

Changes to the URBS policy and/or the AppCL may require that the joint server be periodically rebuilt, i.e., changes to AppCL may still impact SCL. As long as those changes don't alter the signatures of the various methods/protocols that tools utilize, there should be no impact on the tool code. Basically, the dimension of evolvability allows the easy addition of new tools or new users utilizing existing tools. Database integration of AppCL is the same as CP2. However, from a performance perspective, since all security requests are processed by a joint server, there is the potential that the server will become a bottleneck as the throughput of the system increases, i.e., with more tools, or more users utilizing existing tools.

Variant CP4, as shown in Figure 4, is presented as a means to alleviate the remaining consistency, assurance, and performance concerns of CP3. Variant CP4 is truly a multi-process, multi-leveled, client/server architecture. In this case, each TCL is a client to an SCL server that provides security for the entire AppCL, i.e., the SCLi's are replicated. Each SCL, in turn, is a client to the shared AppCL. Like CP2, SCLi's separation from AppCL negates UCLA and GEA as appropriate solutions.

The relationship between each TCLi.j and its respective SCLi acquires the advantages of CP3 with respect to: the independence of the tool code from SCL (and AppCL); the ability to add new tools; and, the lack of impact of changes to SCL (and AppCL) on the tool code. The multiple SCL servers to the TCL clients also alleviate a level of performance concerns from CP3, allowing more SCLs to be added as more tools (and hence, more users) need to be served. Consistency and assurance in CP4 maintain the benefits of CP3 over the other two variants: each SCL has the entire URBS policy/enforcement, so any changes to the policy can be made and replicated. CP4 still may have performance bottlenecks with respect to access to AppCL. But those bottlenecks have now been delineated from the SCLs, and can be handled by replacing AppCL by a distributed object-oriented class library with database support.

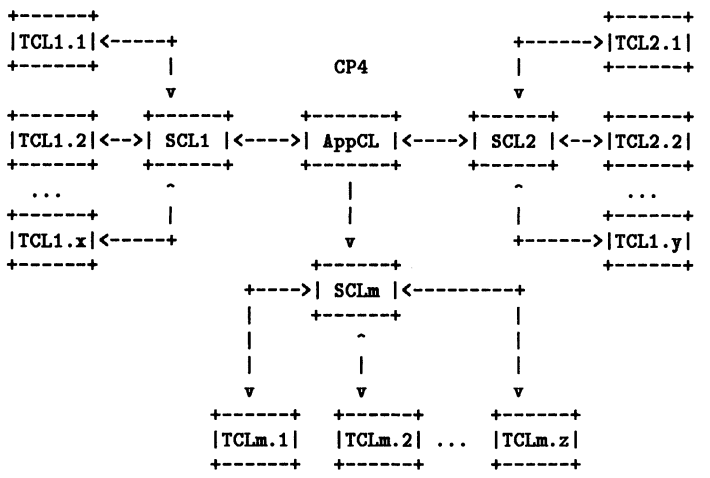

Figure 4 A Multi-Process, Replicated SCL/Shared AppCL Approach.

3.3 Critiquing the Architectural Variants

This sections summarizes the evaluative statements for the six variants into a cohesive discussion that clearly compares and contrasts their capabilities. Our first critique is based on the location and structure of the URBS policy/enforcement within each variant, as shown in Table 1. This is important from a consistency and assurance perspective. In LS1, CP3, and CP4, the entire policy/enforcement is present and captured within SCL (replicated in CP4). In

Table 1 Critiquing Security Policy Location and Structure.

LS1, CP3, CP4	Full/Entire Policy
LS2, CP1, CP2	Partial-Distributed Across Tools
Assessment	Key is Modularity of Security Policy

LS2, CP1, and CP2, the policy is partially captured, to the level required by the tool/TCL. From a consistency perspective, whenever the URBS policy changes, there must be assurance that the policy is still enforced by all existing tools. The centralized nature of LS1, CP3, and CP4, lends itself to a maintenance of the assurance after the change. In the case of LS2, CP1, and CP2, the tools/TCLs must be recompiled to insure that all SCLs are updated. Also, since the policy is spread across multiple SC/AppC pairs (in LS2) or is unique to each process (in CP1 and CP2), there is a chance that inconsistencies can arise that impact on assurance, if all recompilations are not carefully performed.

Our second critique, shown in Table 2, involves the impact of changes on

each variant when either the security policy or application classes are changed. For LS1, LS2, CP3, and CP4, as long as accepted object-oriented design techniques (abstraction, representation independence, etc.) have been followed, it should only be necessary to recompile SCLs and/or AppCLs; there should be no impact on tools/TCL. In fact, depending on the actual enforcement approach

Table 2 Critiquing Changes to Policy or Application

LS1, LS2, CP3, CP4	Recompile Tools Only
CP1, CP2	Rebuild/Change Code Possible Since SCL Linked with TCL
Assessment	Understand Change Potential

(UCLA, GEA, or other), two situations might occur: when the security policy changes, SCL or SCL/AppCL may need recompilation; and, when some application classes change, AppCL or AppCL/SCL may need recompilation. Both situations are dependent on the interrelation of the enforcement approach to the application classes. For other variants: when the policy changes, CP1 and CP2 must be rebuilt, since SCL is within the same process/client as the tool/TCL; when some application classes change, each tool/TCL in CP1 that uses the subset that has changed must be recompiled. CP2 behaves in a similar fashion to CP3 and CP4 for changes to the AppCL.

A third critique involves the utility of our existing enforcement mechanism approaches (UCLA and GEA) for the architectural variants. As currently designed, both UCLA and GEA are tightly coupled to AppCL. That is, it would be difficult to cleanly and completely separate out the SCL from the AppCL. This being the case, it is apparent that some variants are more conducive to the two approaches than others. Namely, LS1, LS2, CP1, and CP3, can all function with either UCLA or GEA as SCL, since SCL is linked to AppCL. On the other hand, neither CP2 nor CP4 can support UCLA and GEA for the AppCL, without changes to UCLA and GEA that decisively separate the security policy/enforcement from the application class library. It will be necessary to either rework UCLA/GEA, or design new variants to support CP2/CP4.

Our final critique, shown in Table 3, focuses on the case when database interactions are required from the AppCL to a persistent store. LS1, CP2, CP3, and CP4 all separate AppCL from the tools/TCL, meaning that a persistent store can be easily supported. LS2 and CP1 have problems, since each approach utilizes a partial AppCL, for only those classes that are needed by each tool/TCL. Thus, for LS2 and CP1, if database access was to occur, it would likely require that the tools interact to synchronize their requests, which raises many major roadblocks. From a performance perspective, all but CP4 have potential bottlenecks at either the SCL, AppCL, or both. CP4 offers the best solution, and if

needed, the AppCL can be expanded to a distributed object-oriented database to satisfy increases in either tools or users.

Table 3 Critiquing Security Policy Location and Structure.

LS1, CP3, CP4	Full/Entire Policy
LS2, CP1, CP2	Partial-Distributed Across Tools
Assessment	Key is Modularity of Security Policy

Finally, based on Tables 1, 2, and 3, we can compare/contrast the capabilities of the variants, as given in Table 4. In Table 4, LS1 has a definite edge over LS2, with respect to attaining assurance/consistency and supporting persistence, since LS1 is very central in nature with one copy of AppC and SCL. However, the distributed nature of AppC and SCL in LS2 gives it an edge when security policy changes occur. In Table 4, CP3 and CP4 are superior and comparable. From assurance/consistency and security policy evolution perspectives, both CP1 and CP2 suffer from partially replicated/distributed SCL and the interactions between tools and the SCL. The partial replication of AppCL hinders CP1 regarding persistency support. CP2 is comparable to CP3 and CP4 since AppCL is not directly linked to TCL nor SCL.

Table 4 Comparing Communication Process Variants.

	Assurance/ Consistency	Security Policy Evolution	Persistency Support
LS1	Superior		Superior
LS2		Superior	
CP1	Prob. - SCL is Partially Replicated & Distributed	Major Changes Possible Due to Links of TCL & SCL	AppCL Part. & Replicat.
CP2			Superior
CP3 & CP4	Superior	Superior	Superior

4 CONCLUDING REMARKS AND FUTURE WORK

Consistency and assurance for object-oriented systems is critical, since it is their nature to evolve and change over time. When both the application class library and the URBS policy are dynamic, those changes have the potential

to significantly impact on the application's tools, which in turn, impacts on actual users. The emerging discipline of software architectures can be utilized to examine alternative architectural variants for the tools, URBS policy, and application class library. Three variants that we have presented rank comparably: LS1 - a layered system with a shared, URBS policy/enforcement and application class library that is utilized by multiple application tools; CP3 a client/server solution where each tool is a client to a server that consists of a joint process containing the URBS policy/enforcement and application class library; CP4 a multi-level, client server solution where each tool is a client, the URBS policy/enforcement is replicated as a server, and the application class library has its own independent server. Of the three, CP4 lends itself to most easily evolving from a centralized to a distributed object-oriented database.

REFERENCES

[1] S. Demurjian and T.C. Ting, "The Factors that Influence Apropos Security Approaches for the Object-Oriented Paradigm", *Workshops in Computing*, Springer-Verlag, 1994.

[2] S. Demurjian, T.C. Ting, and M.-Y. Hu, "Security for Object-Oriented Databases, Systems, and Applications", in *Progress in Object-Oriented Databases*, Prater (ed.), Ablex, 1997.

[3] S. Demurjian, T.C. Ting, M. Price, and M.-Y. Hu, "Extensible and Reusable Role-Based Object-Oriented Security", in *Database Security, X: Status and Prospects*, Spooner, Samarati, and Sandhu (eds.), Chapman Hall, 1997.

[4] M.-Y. Hu, S. Demurjian, and T.C. Ting, "Unifying Structural and Security Modeling and Analyses in the ADAM Object-Oriented Design Environment", in *Database Security, VIII: Status and Prospects*, Biskup, Landwehr, and Morgenstern (eds.), Elsevier Science, 1994.

[5] D. Needham, et al., " ADAM: A Language-Independent, Object-Oriented, Design Environment for Modeling Inheritance and Relationship Variants in Ada 95, C++, and Eiffel", *Proc. of 1996 TriAda Conf.*, Philadelphia, PA, December 1996.

[6] M. Shaw and D. Garlan, "Software Architecture: Perspectives on an Emerging Discipline", Prentice-Hall, 1996.

5 BIOGRAPHY

Steven A. Demurjian is an Associate Professor of CS&E, and in interested in object-oriented design, security, and reuse. T.C. Ting is a Professor of CS&E, and is interested in security, networks, and engineering/design databases. Major John A. Reisner is in the Air Force and is pursuing his doctorate at UConn.