The Impact of Certification Criteria on Integrated COTS-Based Systems

M. Kelkar R. Perry R. Gamble A. Walvekar

Department of Mathematical & Computer Science
University of Tulsa
Tulsa, OK 74104
gamble@utulsa.edu

Abstract

While COTS products can be made secure and reliable within a individual domains, they may introduce security vulnerabilities when integrated with other components due to different security expectations. These problematic interactions within an integrated system can be hidden among the multiple, contributing policy types. Furthermore, security certification criteria governing the integrated system can introduce conflicts with local component policies. Security policies and certification criteria lack a common representation. Security policies use various formats and levels of granularity without comparable attributes. Certification criteria are often text-based checklists. We outline a policy configuration model to represent security policies in a format which can manifest conflicting properties across policy specifications. The model defines security policies according to fundamental attributes of property assertions, observable behaviors, mechanisms, constraints, communication and interaction expectations, dependencies on other policies, system configuration, and component state. We extend model expression concepts to incorporate requirements based on common certification criteria.

1. Introduction

Technological advances and shrinking budgets have resulted in a greater need for software consolidation across distributed infrastructures with seamless, secure information flow. Software components (i.e. legacy systems, COTS products, and third-party software) are being partnered, but may not have been initially created or certified to securely interact with one another. Thus, their integration will neither be as flawless nor as effortless as expected. In fact, it will likely lead to security vulnerabilities. Examples include: (1) component reliance on different access control standards, even within the same organization [1], and (2) data classification distinctions(i.e., unclassified, restricted, confidential, secret, top secret).

Security certification assesses the security posture of a software system to determine and verify compliance with diverse, pre-specified security requirements. The process examines security vulnerabilities that can occur when defined security policies contradict one another or violate certification criteria. It also investigates the risk or security threat level involved due to these vulnerabilities. If performed properly, certification should increase the level of confidence in the security of distributed systems [2, 3].

COTS components do not necessarily expose all security policy information needed to make an exact conflict assessment regarding certification criteria. To compensate, security properties should be defined from a COTS interaction perspective and exposed by the component interface [4]. These property definitions help in assessing security interaction policy conflicts and their impact in overall system security certification.

Security certification is expected to ensure that the procured COTS components fulfill certain security requirements without conflict among component policies [5]. Certification similarly should ensure that component interactions do not contradict the globally defined system policies that cause a violation of certification criteria [6]. One drawback to these expectations is that security certification documents [2, 3, 7] do not explicitly specify criteria specific to integrated COTS based systems. This absence undoubtedly creates the need for restructuring already existing component-based certification criteria when the certification task is at hand. Such restructuring would consider the effect of certification criteria on COTS interactions and comparatively on COTS security policy compatibility checking. However, consistent interpretation and application of criteria to

diverse systems is not guaranteed. One distillation approach toward achieving consistency is being addressed in companion research [8].

In multi-component systems, security interaction policy conflicts can be categorized into direct and indirect types. Direct conflicts can be detected by identifying value-mismatches under same policy definitions and assertions, such as identity versus credential based authentication, one-of versus interactive access control, etc [4]. Unfortunately, conflict detection and assessment process for indirect conflicts is not as straightforward as the direct conflicts. Indirect conflicts appear across different policies and assertions whose dependence may be influenced on requirements handed down from certification criteria. Indirect conflicts detection needs assessment of interdomain mappings, which can be achieved with complex models. Thus, the most immediate need in security certification is defining a policy model which would allow for comparisons across different policy types to determine non-compliance with certification criteria.

In this paper, we devise a uniform security policy model for comparison across components. The model spans multiple security policy types and manifests indirect conflicts across distinct policies caused by the certification criteria requirements. Formulated in UML, the model details policy representations and constraints that are specific to COTS systems.

2. Policy Modeling

Security certification of integrated systems composed of COTS components is necessary to ensure that the procured components, which match the desired functionality, are already certified in isolation as an atomic entity [5]. Apart from this, the certification should include the security property compatibility checking between interacting components and effect of external entities (global properties) [6, 9]. To address the pressing security issues associated with software and its development, the department of defense has mandated the definition of certification criteria and processes to accredit information systems for compliance with security policies [10]. Failure to attain this may result in incomplete certification and lead to unidentified vulnerabilities. Such vulnerabilities include poor encryption and backdoors, which can create exploitable configurations that generate cascading problems through both public and private sectors.

Ensuring individual component security is not enough to determine the security of a system of components [6, 11, 12]. If each component is examined separately, ignoring the interrelationships among components, security can be violated. Approaches for addressing policy conflicts provide examples of security mismatches between two interacting components, but cannot detect security vulnerabilities at implementation [1, 13]

Defining COTS component security properties and mechanisms in a uniform manner is a necessary step towards building the evaluation criteria for standalone products [14]. The Common Criteria (CC) is a standard defining security requirements and evaluating assurance levels of COTS components [14]. The CC does not evaluate security properties for system of systems. Some approaches objectively perform a policy model comparison, but do not provide the framework to detect mismatches which cause security vulnerabilities.

Khan and Han [6] expose trust-related attributes of one component for capture by another, to determine property compatibility. A *compositional security contract* (CsC) contains observable security properties at the atomic, composition, and global levels. This approach identifies a limited number of security characteristics from the CC and values that lead to interoperability conflicts. The template provides one of the few uniform representations of a security profile with respect to these characteristics. When a mismatch is found, the contract is discarded.

3. Motivating Example

The need for uniform modeling and analysis can be shown by an example. Figure 1 illustrates two COTS components, A and C, which interact through a series of entrusted tasks. In order for A to perform task Z, task Y must first be delegated to C. When task Y has executed, Z executes.

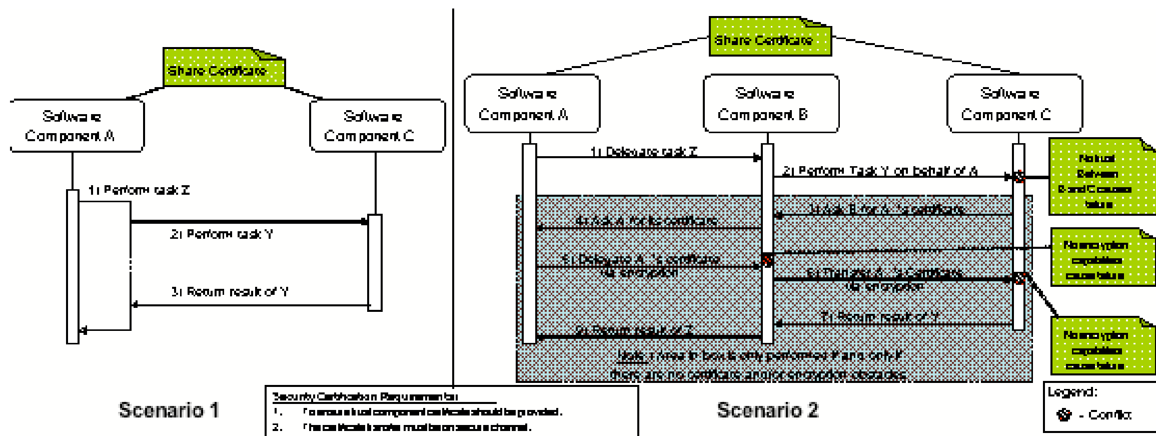

Figure 1. COTS Security Policy Interaction

We define common security concepts [15]:

- *Trust* – Trust is relying on the system to work in a pre-specified manner or have confidence in the system that abides by a set of predefined policies.
- *Certificate* – A certificate digitally signs collections of information such as the ID of a user and its public key. The certificate can verify that a public key belongs to that party.
- *Certificate Sharing* – Sharing a certificate happens when a certification authority (CA) sends the same CA certificate to communicating parties, allowing the communicating parties to establish trust.
- *Certificate Delegation* – Certificate delegation is the process of passing a certificate to another party, empowering the recipient to perform a task on behalf of the owner of the certificate.
- *Encryption* – Encryption obscures sensitive information to make it unreadable without special knowledge.
- *Secure Channel* – A secure channel provides a means of transferring without risk of interception or tampering. Encryption is one methodology to provide secure channel.

In Figure 1, components A and C share a certificate that allows them to trust one another. This avoids the need for certificate delegation to establish trust. It also avoids the need for encryption to build a secure channel between A and C. Though the local policies of each component do not have encryption capabilities, the overall security posture of the system is not affected. As a result, the integrated system complies with the two explicitly stated certification criteria leading to certification of the system.

Problems arise when a component B must be inserted as a proxy of A (Figure 1 – Scenario 2). In the newly configured system, A now assigns task Z to B, requiring B to delegate task Y to C. This causes A to delegate tasks to B and eliminates direct access to C.

Because of mandates on the security certification process, this small change causes four major issues.

1. *Recertifying the System* – Any modifications in the certified system leads to recertification of the security of the system [2].
2. *Ensuring Trust* – For recertification, the certification criteria should be fulfilled. Thus, establishing trust between the interacting components to satisfy the first certification criteria is needed.
3. *Evaluating Security Policy Conflicts* – Conflict detection purveys the compliance of locally defined policies of interacting components.
4. *Evaluating the Effect of the Certification Requirements* – Apart from direct policy conflicts, requirements generated by certification criteria can cause cross-policy dependencies between authentication and data protection policies. These dependencies can lead to indirect conflicts.

A policy model is the best mechanism to address these issues. The model should be uniform and comparable such that the direct policy conflicts can be easily identified. Therefore, the defined model should integrate the requirements and allow direct comparison with the locally defined policies to demonstrate the influence of certification (or recertification) on the system. The model should also make any policy certification requirement conflicts apparent, especially those which might otherwise not be detected through basic design analysis.

The well-developed policy model should be able to show that A and C do not share a certificate with B. If trust is required, it should be established with the certificate delegation process. However, given the requirements, this delegation results in the need for

encryption. Certificate delegation and encryption status are properties of distinct security policies, namely authentication and data protection. Thus, the inter-policy dependencies must be clearly understood. Referring to the certification criteria requirements shown in Figure 1, we arrive at the following conclusions.

- The system abides by the first certification criteria as certificate delegation provides the mechanism for all components to share trust.
- The encryption incapability violates the second certification criteria, failing the recertification.

In the current state of practice, security policy models are restricted to policy matching within one policy type [4, 6]. They are not supportive enough for inter-component policy comparisons to detect either direct local policy mismatches or to examine the effect of certification criteria on COTS policy interactions. In our approach, we have developed a policy configuration model that provides a foundation to address the above issues.

4 Policy Configuration Model and Descriptors

Security policies address different overall concepts including goals, problems, system, critical usage and domain of the application. The six common policy types on which we founded our model are: authentication, authorization, data protection, audit, availability and non-repudiation. Authentication is the process of verifying the digital identity of a component, as illustrated with certificate delegation in Section 3. Authorization protects component data by only allowing that data to be used by those subjects (users or components) that have been granted the authority to do so. Data protection represents a component's process of keeping private information out of public view, as illustrated via encryption in Section 3. Audit is used as a process to verify that security requirements have been met, and even to identify areas in need of improvement. Availability is the accessibility of authorized component resources, in an uninterrupted and timely manner. Non-repudiation tracks and verifies that data has been sent and received by designated components. These six policy types cover the main areas of security policies and thus allow the model to be used to detect a broad range of security problems that exist in integrated networks [15].

We separate relevant policy and component information across distinct concerns, introducing eight descriptors to structure the policies. The first five descriptors are static descriptors, instantiated at design time and assumed to be maintained throughout component execution. The last three are run-time descriptors, whose contents vary according to the configuration of the system, how the component communicates with is interaction partners, and the state of the component policy variables. We define each descriptor, using the authentication policy type as an example.

1. **Security Policy Assertions** – Assertions indicate *the policies that a component adheres to locally*. The absence of a policy is a policy in itself and must be accounted for [10, 15]. For authentication, the security policy assertions are authentication data type, interaction requirements, response, and data reuse.
2. **Observable Behaviors** – This descriptor projects *how the policies dictate component behavior*. What is expected of the component's behavior, given the policy attributes and constraints should be detailed to determine if conflicting component behaviors or expectations contribute to security policy non-compliance. The behaviors for authentication are:
   a. Authentication data type
      i. *Identity Based* – Use of username/password for authentication.
      ii. *Credential Based* – Use of certificate, PGP (Pretty Good Privacy), etc.
   b. Authentication interactivity
      i. *One-time* – Credentials are accepted or produced one time, e.g. single sign-on.
      ii. *Interactive* – Additional credentials may be required, e.g. challenge response.
   c. Authentication data interpretation (response)
      i. *Abduction* – Replies back and expects reply if the authentication data is valid and access is allowed.
      ii. *Deduction* - Replies back and expects a reply authentication failed.
   d. Authentication data reuse
      i. *Single-use* – Authentication data is used only one time, e.g., one time passwords.
      ii. *Multiple* – Same authentication data is used for re-authentication, e.g. double authentication.
3. **Mechanisms -** This descriptor represents *how the policy is deployed or implemented*. There are different mechanisms to deploy policy functions and to implement constraints in behavior and communication. The software mechanisms used to implement authentication include username-password pair, digital certificates, PGP, etc. The hardware types of authentications mechanisms

include biometric devices (finger prints, eye), smart cards, scanning, etc.

4. **Constraints -** Constraints are *external restrictions on a policy.* These constraints may apply to combinations of policy attributes, timing, and other restrictions on policy usage within the DDI system. Authentication may have a location constraint that makes it impossible to login from outside a firewall. A temporal constraint may be needed for re-authentication after idle periods. A numeric constraint can be the number of times failed logins are allowed.
5. **Dependencies -** This descriptor provides common notions of interdependence among different security policies. It is used to better segregate attributes of different policies across policy types. Often, there are anecdotal or commonly accepted connections between different security policy types. These are part of the dependency descriptor as well. As discussed in Section 3, the expression of dependency can facilitate identifying the inter-policy conflicts. Authentication can be dependent on audit, when logging and auditing are needed to show successful or unsuccessful attempts at authentication to detect intrusion.
6. **Configuration -** This descriptor details a component's communication partners inside the DDI system. Assessing policies across disconnected components is not usually necessary. However, there is a need to understand the path of interaction among multiple components and the potential reachability of a detected vulnerability.
7. **Security Communication -** This descriptor represents *policy properties that affect interaction*. How the component communicates its policies through an exposed interface or set of exposed interfaces must be expressed. For authentication, we include the following set of communication methods:
   - *User/component registration* – Providing user information
   - *Authentication request* – Supplying authentication data
   - *Authentication response* – Allowed/deny
   - *User-identity binding* – Binding to a role or a user
   - *Security attributes initialization* - Providing access rights to the user
8. **State -** This descriptor defines the state of the component with respect to relevant policy values. Some state variables may dictate policy expectations or invoke policy constraints at the current point in time, or against some future prediction.

Both static and run-time descriptors can be used in a policy configuration model. For example, a security assertion descriptor defines the need for software backups at regular time intervals, ensuring availability in case of a software crash. The observable behavior descriptor can show the two possibilities of a backup policy, which can be implemented locally, in a remote location, or on a centralized network node. If the system is backed up on a remote server, it is more secure since it can be recovered even if the system crashes. The backup policy is dependent on the data protection policy that can be represented by a dependency descriptor. The recovery procedure of the centralized backup should involve encrypting backup data as it is transferred over the network. If the system cannot provide encryption, which can be represented by an observable descriptor of the data protection policy, then this requirement is violated.

The descriptors lead directly to a UML model with constraints and certification requirements embodied in the associated Object Constraint Language (OCL). With help of the OCL, the UML model can representg the certification criteria in a comparable fashion to facilitate the compliance assessment with locally defined component policies and DDI system interaction expectations. Thus, the UML model represents a reusable artifact for the expression of security requirements beyond the component policies.

Figure 2 shows the generic policy model with partial information from authentication and data protection policies only. There are multiple options to represent the policy models in UML. One alternative is to define stereotypes for each model type. This approach does not lend itself to each comparison across descriptors. Another choice is integrating the policies within descriptors. This is a drastic change in clarity and compression of the model without removing necessary information.

We describe select classes in Figure 2 in more detail. The *Security Policy Assertions* class conveys the existence of authentication and encryption policy capabilities. Different component policies for *Observable Behavior* define the attributes of this class with identity-based or credential-based values for authentication and asymmetric or symmetric values for encryption. The *Mechanism* class holds information about the tangible way with which authentication and encryption policies are put into place. For example, authentication policy can be implemented using user-name password pair or use of certificates, whereas the encryption policy can be implemented using

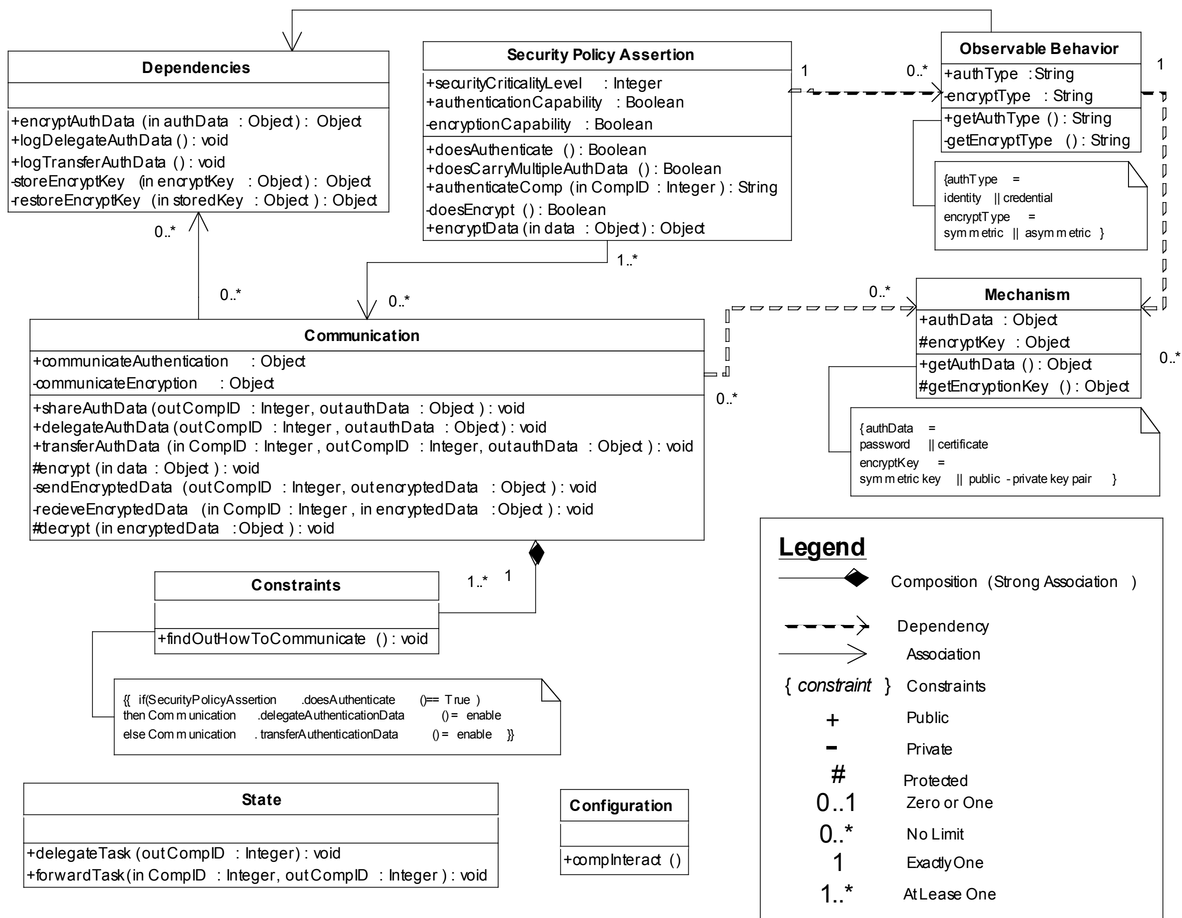

Figure 2. Generic policy configuration model

public/private keys or both. All possible values of the three class attributes are further described in the OCL.

Communication is used with *Constraints* because communication must abide by those constraints. For example, delegation of authentication data is possible only if authentication data exists or otherwise authentication data delegated by some other component can be forwarded using *transfer_authdata* type of communication. The detailed rules of the constraints are described in OCL. There can be additional constraints related to the encryption policy, but they are omitted in this diagram. A more detailed representation of the *Configuration* and *State* descriptors appears in Figure 3 and Figure 4.

5. Security Conflict Detection

In multi-component systems, security interaction policy conflicts can be categorized into two types. Direct conflicts are manifested by contradictions across the same policy type. For example, if two components have distinct values (e.g., identity and credential) for the *authentication_type* attribute in *Observable Behavior* in Figure 3, then these components are unable to exchange their authentication data, leading to a direct conflict. Indirect conflicts are shown as failures to fulfill system policies or security certification requirements due to contradicting component interactions involving multiple policy types. The process to identify indirect conflicts is not as straightforward as that for the direct conflict detection and is explained with the help of our example from Section 3. Further research in process development is needed to encompass all policy types.

The instantiated policies for components A and C are shown in Figure 3. The *Observable Behavior* and *Mechanism* classes contain exactly the same values, indicating no direct conflicts. The security certification

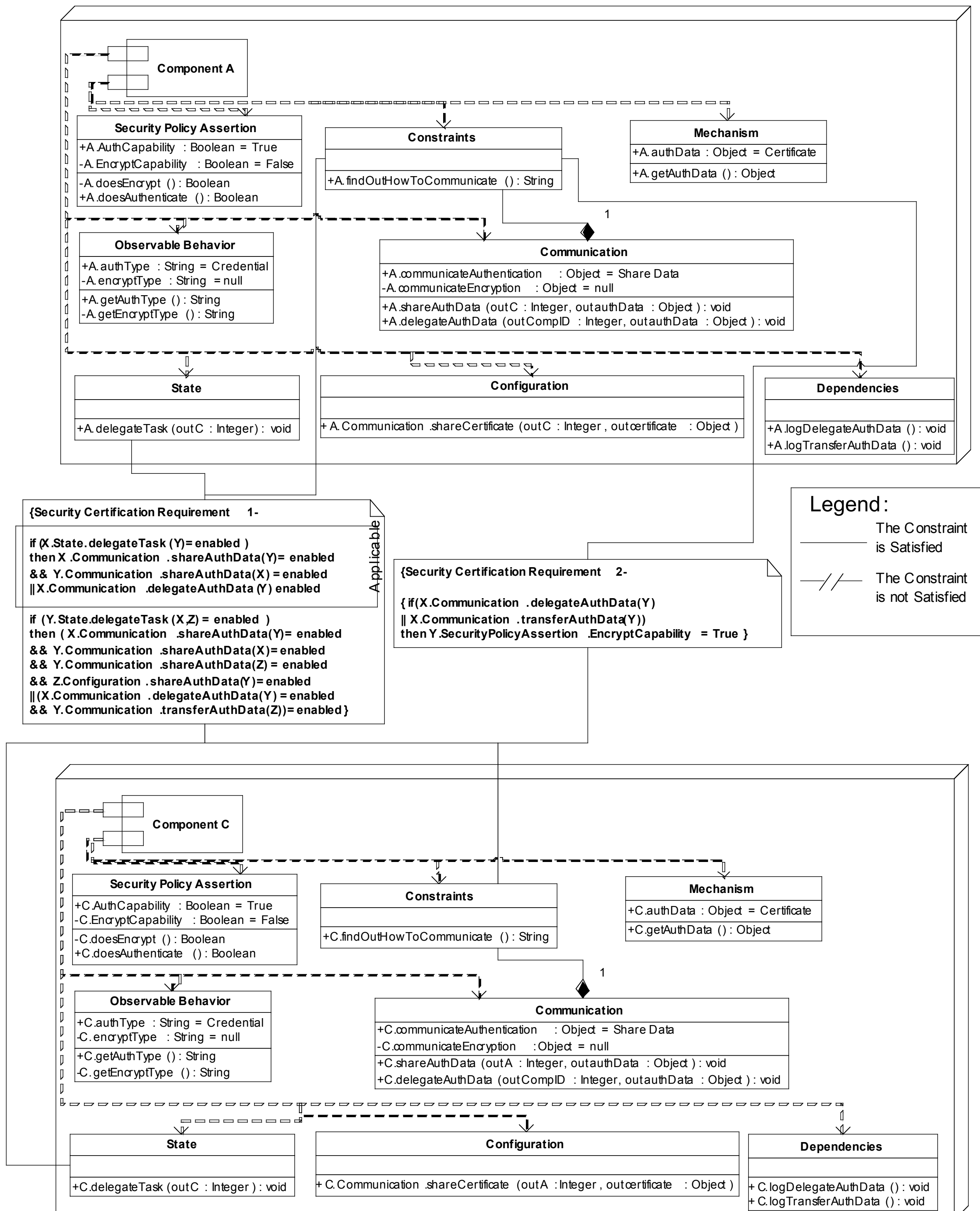

Figure 3. Example scenario 1 - policy configuration model

requirements are represented as constraints applied to the run-time behavior (*State*) and the *Communication* of the system. *State* contains a method showing the requirement to delegate a task from A to C. Thus, only one part of the first requirement is applicable for this system; namely, if a component delegates a task to another component, then either both components should share a certificate or the component who delegates the task should also delegate its certificate to establish trust. In our example, A and C share a certificate which is shown by the *Configuration* in Figure 3. Thus, the first requirement is complied with because of the *shareCertificate(A,C)* method in the *Configuration* class. This avoids the need for certificate delegation and thus, encryption. By default the system complies with the second requirement and can be certified.

Insertion of Proxy B into the system alters its configuration, as discussed in Section 3. Since the security posture of A and C is not changed, their static descriptors remain the same. However, the *Communication* descriptor must now reflect the current structure of the interaction which is dictated by the change to *Configuration* (not shown). The *State* descriptor also changes as a result of inserting B. Component B's static descriptors are same as A and C, except B does not carry its own certificate for authentication. Figure 4 shows the modified classes for *Communication* and *State*. The state of the current system configuration and the first security certification requirement is interpreted as:

- The state of A shows that it delegates its task to B. According to the first certification requirement, it should either share a certificate with B or it should be able to delegate its own certificate to establish trust.
- The state of B shows that it delegates a task to C on behalf of A. According to the first certification requirement, B should either share a certificate with C or it should be able to transfer A's certificate to C to establish trust, since B is a proxy.

The *Communication* class of components A, B and C fulfills the requirement because:

- A can delegate its certificate to B
- B can transfer A's certificate to C

Thus, the first security certification requirement is satisfied. The second certification requirement states that if a delegation or transfer of certificate exists, (authentication data) then the certificate should be encrypted. The attributes of the *Security Policy Assertion* class of components A, B and C show that the components *do not* have encryption capability. This requirement is violated by an indirect policy conflict. The indirect conflict is manifested by the representations of the policies within the structured model. Figure 4 denotes that Security Certification Requirement 2 is unfulfilled.

6. Conclusion

In this paper, we have uniformly modeled security policy types from a COTS interaction perspective using our Policy Configuration Model. The model is represented by descriptors using OCL. This model aids comparison across components to detect direct (within one policy) and indirect (effect of certification criteria) type of conflicts. Detecting direct policy conflicts is straightforward and involves detection of attribute mismatches in the static state descriptor classes. We used COTS-specific certification requirements distilled from generic security certification criteria to detect indirect conflicts.

We are in the process of accumulating, classifying, and organizing security policy information from a wide variety of reliable sources to establish common artifacts among COTS based system component security policies. We are working towards standardizing a process of distributing information within the policy configuration model. By using UML, we can employ supporting tools to determine compliance with policy constraints. A survey of available UML tools availability for defining OCL constraints is being performed to determine the most feasible tool for constructing an analyzable model[16]. Construction of the policy configuration model in the chosen tool would further facilitate its validation. The tool can be used to analyze interacting component policies to determine if conflicts are present.

Acknowledgements. This work is supported in part by a U.S. AFOSR award FA9550-05-1-0374. The U.S. government has certain rights to this publication.

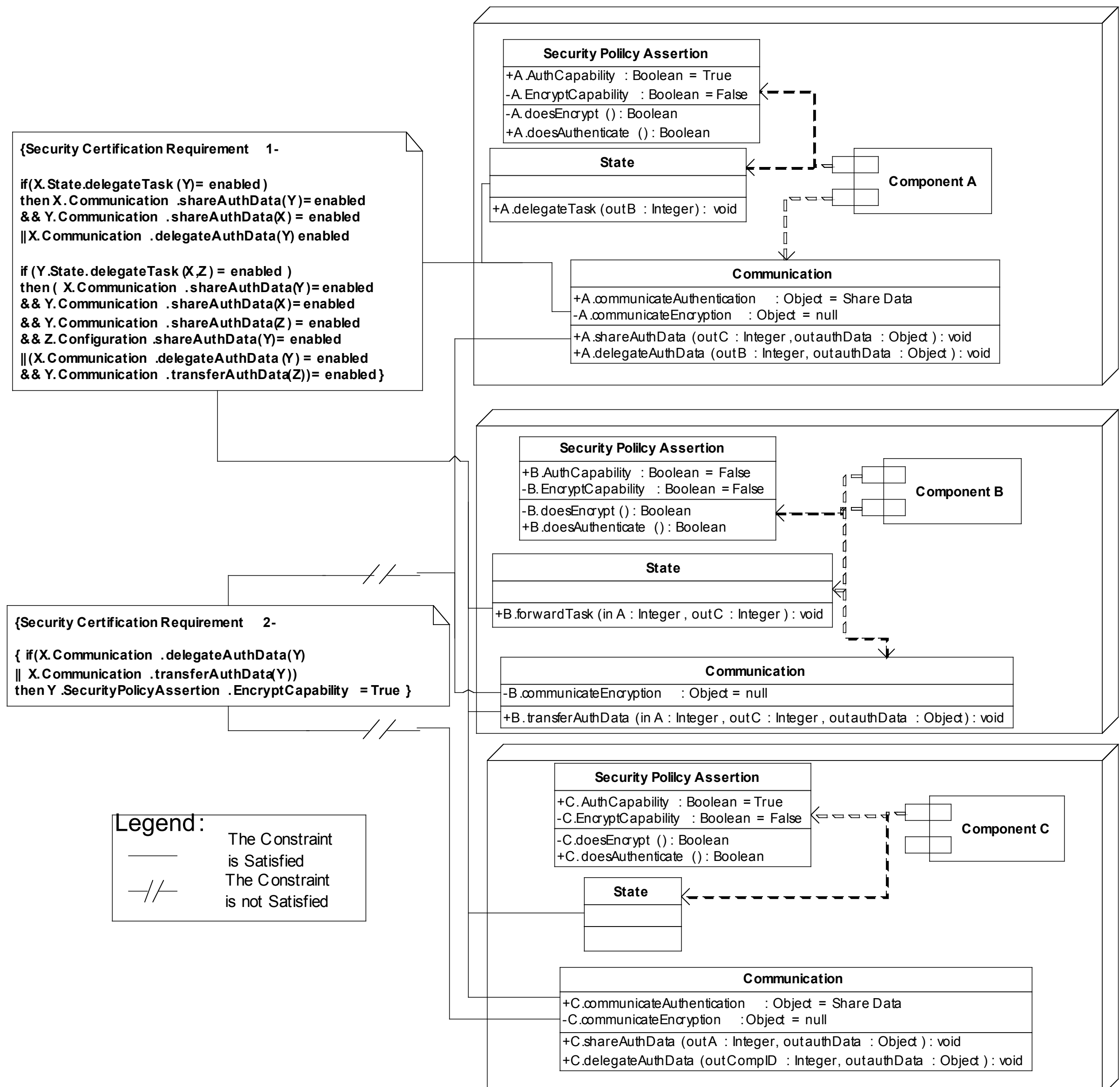

Figure 4. Example scenario 2 - policy configuration model

7. References

[1] S. A. Hissam and D. Plakosh, "COTS in the Real World: A Case Study in Risk Discovery and Repair," Pittsburgh, PA June 1999.

[2] "Department of Defense Information Technology Security Certification and Accreditation Process," July 2000.

[3] R. Ross, "Guide for the Security Certification and Accreditation of Federal Information Systems," May 2004.

[4] M. A. Kelkar, M. Smith, and R. Gamble, "Interaction Partnering Criteria for COTS Components," presented at the International Conference on Software Engineering and Knowledge Engineering, San Francisco, CA, 2006.

[5] J. Boegh, "Certifying Software Component Attributes," *IEEE Software*, vol. 23, pp. 74 - 81, 2006.

[6] K. M. Khan and J. Han, "Deriving Systems Level Security Properties of Component Based Composite Systems," presented at the Australian Software Engineering Conference (ASWEC'05), 2005.

[7] "Common Criteria for Information Technology Security Evaluation," June 2005.

[8] M. A. Kelkar, R. Perry, and R. Gamble, "Certification Distillation Technical Report," University of Tulsa, Software Engineering and Architecture Team, 2006.

[9] H. Mei, J. Lukkien, and J. Muskens, "A Compositional Claim-based Component Certification Procedure," 30th EUROMICRO Conference (EUROMICRO'04), 2004.

[10] J. H. Allen, *CERT Guide to Systems and Network Security Practices*, 2nd ed: Addison Wesley, 2001.

[11] L. Qin and V. Atluri, "Concept-Level Access Control for the Semantic Web," presented at 2003 ACM Workshop on XML Security, Fairfax, VA, 2003.

[12] R. Bhatti, B. Shafiq, E. Bertino, A. Ghafoor, and J. B. Joshi, "X-GTRBAC Admin: A Decentralized Administration Model for Enterprise-Wide Access Control," *ACM Transactions on Information and System Security*, vol. 8, pp. 388-423, 2005.

[13] U. Lindqvist and E. Jonsson, "A Map of Security Risks Associated with Using COTS," *Computers IEEE*, vol. 31, pp. 60-66, 1998.

[14] W. J. Lloyd, "A Common Criteria Based Approach for COTS Component Selection," *Journal of Object Technology*, vol. 4, pp. 27-34, 2005.

[15] C. P. Pfleeger, *Security in Computing*: Prentice-Hall, Inc., 1997.

[16] M. A. Kelkar and R. Gamble, "Establishing a Framework for Security Certification Models Technical Report," University of Tulsa, Software Engineering and Architecture Team, 2006.