Taming Access Control Security: Extending Capabilities using the *Views* Relationship

Marcus E. Markiewicz
e-mail: mem@acm.org

Carlos J. P. Lucena
e-mail: lucena@inf.puc-rio.br

Donald D. Cowan
e-mail: dcowan@csg.uwaterloo.ca
Computer Science Department and Computer Systems Group
University of Waterloo, Canada

PUC-RioInf.MCC19/00 May, 2000

Abstract - The "views" relationship indicates how an object-oriented design can be clearly separated into objects and their corresponding interface. This paper uses the concept of "views" in order to achieve full separation between the application and the security policy in the design and implementation. The result is achieved by providing a model for capabilities using "views" that is richer than the traditional capability model. In addition, a distributed access control model is shown to be effective through the use of Secure Object Communication Channels (SOCCs) to allow for secure connections at the abstract object level. This security is applicable in the e-commerce application domain, bringing security directly to the application abstraction level.
Keywords - Views, capability, subject-oriented programming, reuse, security, e-commerce.

Resumo - O relacionamento "views" indica como um design orientado a objeto pode ser separado em objetos e suas respectivas interfaces. Este artigo usa o conceito de "views" de forma a alcançar a separação completa entre o código de uma aplicação e a política de segurança no seu design e implementação. O resultado é atingido por um modelo proposto para "capabilities" utilizando "views", sendo este mais rico que a abordagem tradicional. Em conseguinte, um modelo de controle de acesso distribuído utilizando "Secure Object Communication Channels" (SOCCs) é demonstrado eficiente ao permitir conexões seguras no nível de abstração de objetos. A segurança alcançada por este modelo é aplicável no domínio de aplicações de "e-commerce", provendo segurança diretamente no nível de abstração dos aplicativos.
Palavras Chave - Views, Capability, subject-oriented programming, reuso, segurança, e-commerce.

Sponsored by IBM Brazil

1 Introduction

Real world entities modeled by objects in object-oriented design and programming have both intrinsic and extrinsic properties. For example, a pen has its physical characteristics such as its size and color. These are intrinsic properties, which cannot be separated from the object. An extrinsic property would be its purchase value. Every buyer might have different valuations of its price, and these values are simply unknown to the evaluators, unless the price is physically associated with the pen.

Extrinsic properties are almost always subjective and context-dependent, which restrains the reuse of objects. If we create versions of the object for each context, we are duplicating code, which is unacceptable. Other solutions involve the modification of the object code such as adding these extrinsic properties through inheritance or including identity when accessing all the objects methods [1]. Designers are forced either to anticipate all the future use of the applications treating all extrinsic information as intrinsic, or to forego the object-oriented style.

The existence of the extrinsic properties has been observed in [2], and the limitation they impose on object-oriented analysis and design methods (OOADM) [3]. Extensions to object-oriented programming called subject-oriented programming have been proposed [2]. Subject-oriented programming [2] proposes a solution for the above mentioned problems by creating views or subjects of objects, allowing the manipulation of their extrinsic behavior. In this paper, we consider the "views" model and abstract design views (ADVs) [4][1]to be conceptually equivalent to the subject-oriented model, although at a different level of abstraction [1].

In object-based systems, capability-based security is a well-known approach to access control, but has several deficiencies such as the hard to trace leak of access to non-authorized parts of applications. In this paper, we will consider the access control of applications to be an extrinsic property that can be easily traceable using ADVs or the subject-oriented model. This approach will allow us to reuse the policy code and also support a full separation of concerns [5].

Some access control decisions require that ADVs (viewing objects) receive or fetch information from an external source. For this matter, ADVs can be used for secure communication, using Secure Object Communication Channels (SOCCs), a concept described in this paper. The use of SOCCs provides security to the transmission, since the data is encrypted and thus

[1]Since after the publication of [4], we have started using ADV and ADO as the acronyms for Abstract Design View and Abstract Design Object and not as Abstract Data View and Abstract Data Object as we did in earlier publications.

protected from eavesdroppers.

The architectures presented in this paper allows any e-commerce or web application to have proper security at the abstract design level. Lightweight SOCC-based secure connections can be used to secure the applications, without any specific platform or operating system policy or proprietary secure communication mechanism. The reusability of the access control code allows the security aspect of the e-commerce applications to keep pace with this fast paced market.

Section 2 outlines the concepts of abstract design objects and abstract design views. Section 3 shows the relationship of ADVs to access control and the security architecture. Section 4 discusses how to model capabilities using ADVs, possible architectures and details on ADV implementation. Section 5 discusses how to model remote information dependent constraints using ADVs. Section 6 argues that the use of ADVs provides a richer model than plain capabilities. Section 7 discusses how to map ADVs to design patterns. Section 8 provides a more concrete example of the model and explains how the access control code can be reused. Finally, in section 9 conclusions and future work are discussed. Please notice that all the design examples are expressed in UML [3], [6].

2 The "*Views*" relationship and Abstract Design Views

The "views" or "views-a" relationship and its realization through Abstract Design Views (ADV) [4] is a concept that promotes loose coupling between modules [7]. It is maintained between two objects, where one object, called the view, monitors or "views" the state of another related object (viewed). The viewed object is unaware of the existence of the viewing object.

The properties of this relationship characterize the static and dynamic constraints, which determine the type of interaction between objects, defining how an action occurring in one object influences another object. This relationship can be modeled in UML by the extension of the UML metamodel, as in [8].

The "views" relationship can be modeled using using ADVs and ADOs. For this purpose, in this paper ADVs are extended in a way that they can relate with another ADVs as they do with an ADO. This extension allows ADVs and ADOs to realize the transitivity of the "views" relationship [8], estabilishing themselves as a lower abstraction model for this relationship. The "viewed" is the ADO, and the "viewer" is the ADV.

2.1 Abstract Design Objects and Abstract Design Views

The Abstract Design View approach was originally motivated by the use of pairs of objects to represent application components and their interface in reusable designs [9], [10]. The application components and interface components are called respectively abstract design objects (ADOs) and abstract design views (ADVs). An ADV is used as an interface (in a very broad sense) for ADOs in designs and provides a "view" of an ADO. We chose to prefix the name of both types of components with the word abstract because in design we are only interested in their behavior, not their implementation. Composition constructors are used to combine ADVs and ADOs to produce more complex designs, and this process has been validated by proof of concept architectures. The approach can be seen as a way of providing language support for the specification and abstraction of inter-object behavior [11].

ADVs have been used to support user interfaces for games and a graph editor [12], to interconnect modules in a user interface design system (UIDS) [13], to support concurrency in a cooperative drawing tool, and to design and implement both a ray-tracer in a distributed environment [14] and a scientific visualization system for the Riemann problem. The VX-REXX [15] system that was built as a research prototype, was motivated by the idea of composing applications in the ADV/ADO style.

ADVcharts, a graphical formalism for representing designs using ADVs, have been tested in the production of several different software designs. ADVcharts have also been used to redesign and reengineer an existing interactive software system [16]. In addition, it was shown in [10] and [17] how ADVs can be used to compose complex applications from simpler ones in a style which is similar to some approaches to component-oriented software development [18] and megaprogramming [19].

2.1.1 Abstract Design Objects

The actions or methods of an ADO can only be called by another ADO or ADV but not by an event that is outside the set of objects. An ADO accesses other ADO methods and properties but this only happens when the ADO is left unprotected. In this way, the ADO can use the other ADO as it wishes, without any constraints.

In the normal case where we wish to protect the ADO, no one has access to an ADO's methods and properties, but its associated ADV. In this case, any external stimuli will have to go through an ADV.

2.1.2 Abstract Design Views

ADVs were originally conceived to act as general event-driven user interfaces and to achieve a separation of concerns by providing a clear separation at the design level between the application as represented by an ADO, and the user interface or ADV.

ADVs support one or more mappings which allow an ADV instance to query or change the state of any associated ADO instance through its action interface. This mapping approach allows controlled access to the state of an ADO instance consistent with the hiding principle [20]. The mapping is specified through an owner variable that represents the associated ADO instance. The strategy used to implement the owner variable mapping is not specified in the design model, but can be specified through design patterns such as the "Observer [21]" when the design is implemented.

The action interface of an ADV instance can be invoked both by input messages and through the usual methods. Input messages can be triggered by external operations or events caused by an input device such as a keyboard, a mouse, or a timer. Thus, the action interface of the ADV extends the ADO interface to include external events. We call these events causal actions, because they represent the root cause of a computation, that is some form of external input.

ADV instances send output messages if their appearance is altered through an input message or a change in state of accompanying ADO instance. An ADV instance uses the owner variable mapping to query the state of an accompanying ADO instance to determine if it has changed and effects the ADV instance. Output messages in terms of a user interface are the display commands which paint various views on the screen.

The relationship between an ADV and an ADO is not symmetric, since several ADV instances could be associated with the same ADO instance in order to provide different views or control functionality. This many-to-one relationship means that each ADV instance must be consistent with the associated ADO instance (vertical consistency), and that all the ADV instance must be consistent with each other (horizontal consistency).

In this paper, the original ADV model [4] is extended in a way that ADV's can relate to each other like ADV's relate to ADO's.

2.2 Communications between ADVs and ADOs

Communication between an ADV instance and an ADO instance occurs when the ADV instance executes synchronous invocations (procedure calls, effectively) to the ADO instance [22]. An ADO instance never generates events that must be handled asynchronously by the associated ADV instance. This approach contrasts with other user interface models, where a component must handle both synchronous and asynchronous invocations from other components. Handling asynchronous invocations is considerably more complicated than handling synchronous invocations since it requires error-prone mechanisms such as signals, interrupts, or callbacks. With an explicit mapping we enforce a one-way communication, and, as a consequence, have fewer interconnections, thus, ensuring that the role and scope of the interface are defined unambiguously. Asynchrony is needed in other models because the "official" locus of control is in the nonuser-interface application. This approach is consistent with "slightly-interactive" programs, which mostly compute but occasionally prompt the user for input. However, in highly interactive applications the actual locus of control is associated with the user. The tension between these two loci of control is the source of the complexity. The ADV model avoids this tension by placing the main locus of control in the ADV. Fundamentally; the ADV model is based on the program waiting for the user rather than the user waiting for the program.

3 Access Control Architectures

In this section, we describe our security model for access control using abstract design views and "views".

In object-based systems, access control is often based on capabilities [23], since capability-based security is a well-known paradigm. In this case, object references are capabilities, and if one has this reference, there is full access to the object. Capabilities can be viewed as the key to a safe. Once one has access to the capability or key, nothing can prevent him or her from opening the safe. Using the views of an object, we show how to enforce policies on capabilities.

One serious deficiency of capabilities is the frequent exchange of object references. In large systems, this exchange makes it extremely difficult to check that parts of an application will not gain access to certain capabilities. Many solutions to this problem depend on domain-based policies or Access Control Lists (ACLs), such as in Java [24]. Domain-based policies do not offer the proper granularity for developers, as only cross-domain accesses are properly distinguished [25]. The use of ACLs does not promote reuse of objects because of the access control constraints are attached to the object. The use of ACLs also suffers from the problem of leaking unprotected references to application parts that are not trustworthy.

Another approach is the use of meta-objects to control the access to references, by attaching themselves to the application's objects [25]. This approach is very effective in controlling leakage of protected references. However, meta-objects are not available as constructs for every object-oriented language (as C++, for example). The use of meta-objects as a security layer also introduces an overhead into the applications. Meta-objects also accumulate as they guard the same reference thereby increasing the complexity and inefficiency of the solution [25].

Traditional security models are centralized, coarse grained and mostly static. These are not suitable for large distributed and autonomous environments such as the Internet. Since roles evolve very quickly, security policy can also change rapidly. There are many solutions to this distribution problem, such as active capabilities [26], that depend on scripts that implement security policies. We believe that the use of "views" is also a approach that is easily distributed, since the access control code always encapsulates the application code, and further code is not dependent on any centralized manager throughout their use. All the dependencies on external information that the constraints might have can be implemented to work in a distributed environment.

4 Modeling Capabilities using the "*Views*" relationship

The use of capabilities with object references can be categorized into three concepts: restriction of access, revocation and expiration. Restriction of access happens when one wants to limit the use of objects, such as to a particular user or group. This type of restriction depends on constraints or rules. Revocation is the ability to stop the use of an object by declaring that the specific use is no longer valid. Only the owner of the objects or someone who has the permission of the owner should be allowed to revoke objects. Expiration is a time constraint or a time limit, such as a valid date for a credit card.

All these new concepts fall under two possible types of constraints: active and passive. The active constraints take a pro-active attitude, by accessing local and remote resources to determine if the constraint should allow access. One such example is a restriction of use based on the user identity. In this case, access will only be allowed when the identity is authenticated. For these constraints, a view of an object can act as an agent that gathers the proper information to make the access decision. On the other hand, passive constraints are totally dependent on external stimuli. It is possible that access

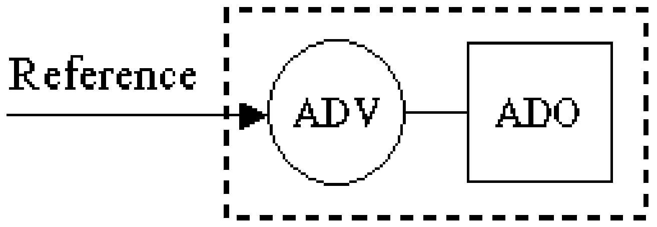

Figure 1: Single ADV approach to access control

control decisions may depend on remote information, that is, information that is possessed by a remote object. This problem is discussed in the next section.

Since capabilities can be understood as references, caution must be taken about who possesses them. Thus, the "viewers" (ADVs) serve as an intermediary between the "viewed" object (ADO) and external references, enforcing the constraints implemented within the ADV. This allows us to isolate the policy code in an ADV, while the application code within an ADO is completely unaware of the security constraint. The ADV is used here as the single point of entry for references to the ADO.

Thus, revocation, restriction of access and expiration can be modeled as constraints in ADVs that are either passive or active ones. When it comes the time to expire or revoke an object, the respective ADV will trigger the change of its accompanying ADO state so that it will become useless or be renewed. As for restriction of access, the constraint ADVs regulate the access to the ADOs, enforcing the desired policy.

Since the ADVs have all the policy code, we are able to reuse this source code, by reusing the policy ADV classes in other designs and contexts. The policies are no longer bound to the context of the application, neither is the application bound to the policy.

Thus, for every object, its many views ("viewers" or ADVs) will enforce policies and constraints. Access to an object is only possible through a "viewer", that can be combined to provide "viewers of viewers", enforcing multiple layered constraints. The next section contains a discussion on the possible ADV and ADO architectures that realize the "views" relationship for access control security.

4.1 Single ADV Architecture

In this approach, we have a single ADV that enforces the access policy for an ADO. This ADV supervises all the access to the methods of the ADO. Figure 1 shows that externally the ADV and ADO look like a single object, and since no one has a reference directly to the ADO, it remains protected.

This approach is clearly easy to implement. However since there are usually multiple constraints, packaging them in one ADV make not be practical. Further it will be difficult to separate and reuse the policy code.

4.2 Multiple ADV Architecture

Another approach is to have an ADV for each constraint. This way, there will be multiple references to multiple ADVs, which will all be connected to the single ADO. This method is shown in Figure 2. ADVs allow this approach because of the vertical and horizontal consistency properties [4].

Each reference will know an ADV, and there will be multiple entry

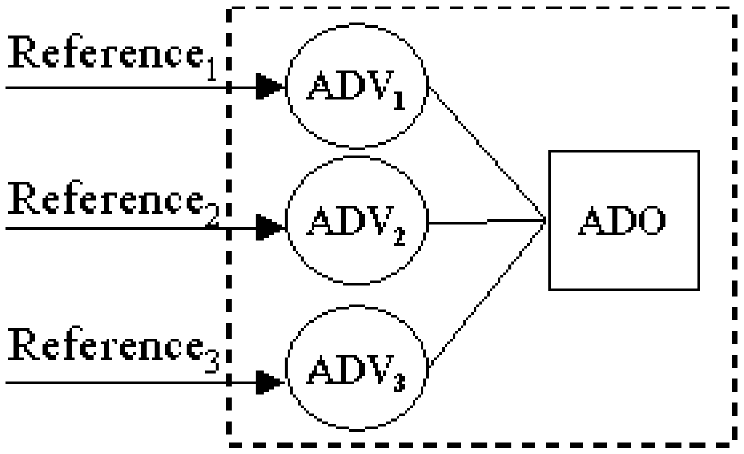

Figure 2: Multiple ADV approach to access control

points for the ADO. Since each constraint is represented by an ADV, reuse of the policy code encapsulated in the ADV is possible even in contexts other than that of the application. This approach introduces another degree of complexity as the use of multiple ADVs and this multiplicity of entry points means external classes must keep multiple references to a single ADO.

4.3 Hybrid Architecture

A hybrid approach that overcomes the disadvantages of both solutions is to have a single ADV that is composed of many ADVs. Therefore, we shall use a composite ADV as described in [4]. This solution provides the benefits of a single entry point and also allows the proper granularity to enforce the constraints and promote reuse of policies. Figure 3 shows how the ADVs relate to each other.

This approach provides an ADV that manages the constraint ADVs,

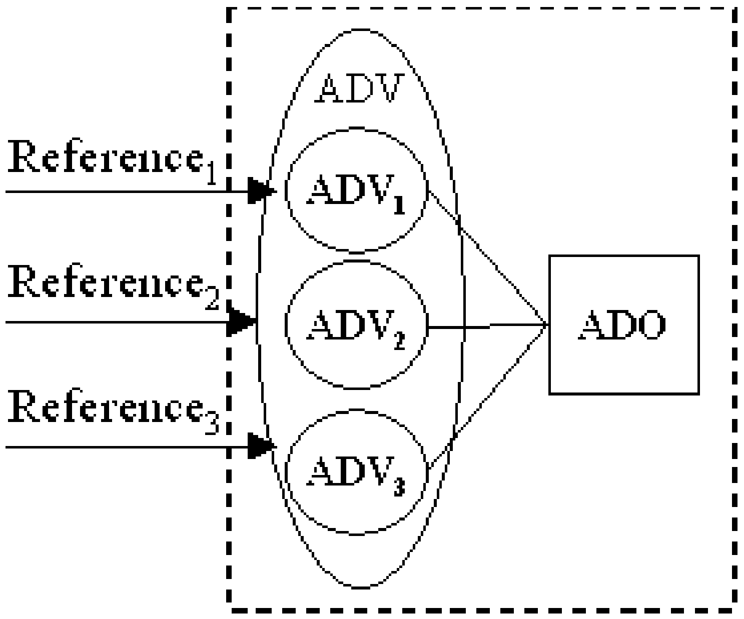

Figure 3: Hybrid ADV approach to access control

and even allows them to change dynamically.

5 Modeling Remote Information Dependent Constraints

Some access control decisions require that ADVs receive or fetch information from an external source. For this matter, ADVs can be used for communication in a client/server architecture, as described in [4]. However, in a highly distributed and autonomous environment like the Internet or Intranets, there are occasions in which objects must take both the server and the client role. Therefore, an ADO might have ADVs acting as clients and others as masters.

Every communication between two entities depends on proper authentication [27]. This authentication can be performed in a one-way or two-way manner. In the classic one-way authentication [27], the server authenticates the client by acknowledging its identity, but the client has no proof whatsoever of the server's authenticity. In two-way authentication [27], the client and the server authenticate themselves, giving proofs of each other identities. The usual proofs for authentication are certificates, like X.509 certificates [28]. These are based on asymmetric encryption key technology [29], [30],

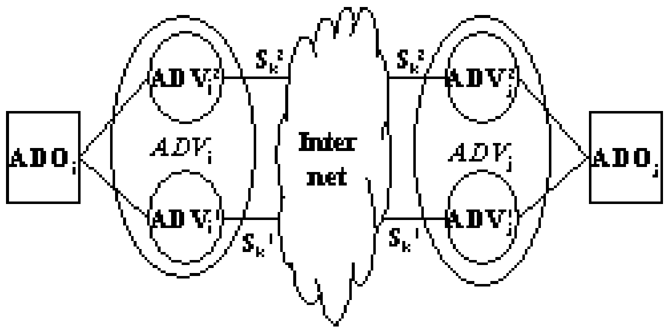

Figure 4: Secure Objects Communication Channel using symmetric encryption

such as RSA [31].

However, in an authentication or access control context, the communication channel between two parties itself must be completely secure, otherwise capabilities might be given away to eavesdroppers. Thus, encryption techniques must also be used in the communication between the two parties.

ADVs can be employed to create secure communication channels. In this case, two or more ADOs that are remote to each other can communicate by creating ADVs that form a channel between them. So that this channel can be secure, the ADVs involved in the communication handshake and agree on an encryption algorithm, and thus create a Secure Object Communication Channel (SOCC).

Figure 4 shows how two distributed ADOs communicate creating SOCCs. In the example, ADO$_i$ wishes to communicate with ADO$_j$. ADO$_i$ creates two ADVs: ADV$_i^1$ and ADV$_i^2$, that will communicate respectively with ADV$_j^1$ and ADV$_j^2$, created by ADO$_j$. Therefore, there will be two SOCCs, that in the example use symmetric encryption keys [29], [30] like DES [32]. Thus, ADV$_i^1$ and ADV$_j^1$ share a secret key S$_k^1$ and ADV$_i^2$ and ADV$_j^2$ share a different secret key S$_k^2$. All the communication between the two ADOs will be performed using encryption. This approach is suitable for the transfer of large amounts of data, since asymmetric encryption is slower than symmetric encryption [29]. If one wishes to use asymmetric encryption [29], [30], the only difference from the example is that each ADV should encrypt the message using the other's ADV public key.

As an example, we present next an outline of the SOCC protocol for two-way authentication. This protocol is similar to the Secure Socket Layer (SSL) protocol, as described in [33] or the Transport Layer Security (TLS) protocol [34], but at an object abstraction level. An implementation of this protocol is possible using one of the many SSL socket libraries available [35]. One should preferably use two-way authentication whenever possible. The use of encryption keys that are used only once for a channel and then discarded (also called ephemeral keys [33]) is also advised for sensitive communication.

1. ADO_1 creates client ADV_1 and ADO_2 creates server ADV_2, that attaches itself to a common knowledge port.
2. The client ADV_1 opens communication with server ADV_2.
3. The client ADV_1 and server ADV_2 begin handshaking.
   (a) The server ADV_2 and client ADV_1 establish their common security protocols [33].
   (b) The client ADV_1 receives a proof of server ADV_2 identity, and verifies it.
   (c) The server ADV_2 receives a proof of client ADV_1 identity, and verifies it.
   (d) The handshake is finished, and from this point on the parties will use the protocols, keys and secrets agreed.
4. The client ADV and server ADV begin sending objects encapsulating the encrypted data, much like packets.

For one-way authentication, the SOCC protocol is the same, but there is no step 3(b).

This way, remote information dependent constraints can be modeled as a local ADV that connects to a remote ADV. These constraints can be either passive or active. The use of SOCCs support secure transmission, since it is encrypted and thus protected from eavesdroppers.

Thus, constraint ADVs are capable of gathering information both locally and remotely, using SOCCs. This allows the access control decision to become as distributed as required by the application. In addition, constraint ADVs can cooperate in decisions and even on revocation or expiration of remote objects, creating composite ADVs that have as many SOCCs as needed.

SOCCs can be implemented as ADVs using Remote Procedure Calls (RPC) to communicate with the ADO. Thus, no architectural distinction is made from the constraint ADV concept and SOCCs.

6 Extending Capabilities

The Abstract Design Views concept supports more uses than simple capabilities. Much like the meta-object approach [25], ADVs allows transitivity of access control and protection against leakage of ADO references. Since we only have access to the ADV of an ADO, this reference can be passed throughout the whole application, and the policy will still be enforced. Therefore, the reference to an ADV can be passed without compromising the security of an ADO.

The use of ADVs prevents the leakage of ADO references only if the properties of ADVs are maintained. That is, no access is allowed to an ADO if not by its ADV or from an ADO. Thus, the code of the ADV must not allow references to the ADO to be exported or given to any external object. As an example, consider determining if a subset of the state of two ADOs is equal. In this case, ADVs must be implemented to provide a view of the two ADO subsets. Then, the results provided by these two ADVs can be compared. Note that in this process no reference to the ADO was leaked. In these cases, the ADV must act as the guardian of the ADO, providing a view of the ADO while protecting it from direct external use.

7 Mapping constraint ADVs using design patterns

The "Views" relationship can be modeled using ADVs and ADOs, in a way that they are roles in this relationship. ADOs poses as the "viewed" object, and ADVs are "viewers."

Since Access Controller ADVs are entry points to the ADO that enforce access control policy, it is possible to model it as a proxy. Constraint ADVs and ADOs can be mapped using the Protection Proxy variant of the proxy pattern, as described in [36]. However, this pattern must be extended. Multiple constraints are applied using the "views" relationship transitivity. To promote the reuse of the constraint ADVs, the "viewer" (ADV) of an object (ADO) will be an access controller. It will use other ADVs as policy enforcers, therefore separating the code of each constraint. The transition of a "views" relationship to a design pattern using ADVs and ADOs is shown in figure 5.

In figure 5, the expanded proxy pattern is represented in a typical dynamic scenario. Note that the pre- and post-conditions are the constraints that perform the access control decision.

Thus, the process of access control security design involves the use of "views" relationships, that are mapped into ADVs and ADOs architectures. The ADVs and ADOs are then modeled using the proxy pattern, for the access control security case. Notice that the constraint ADVs can be reused

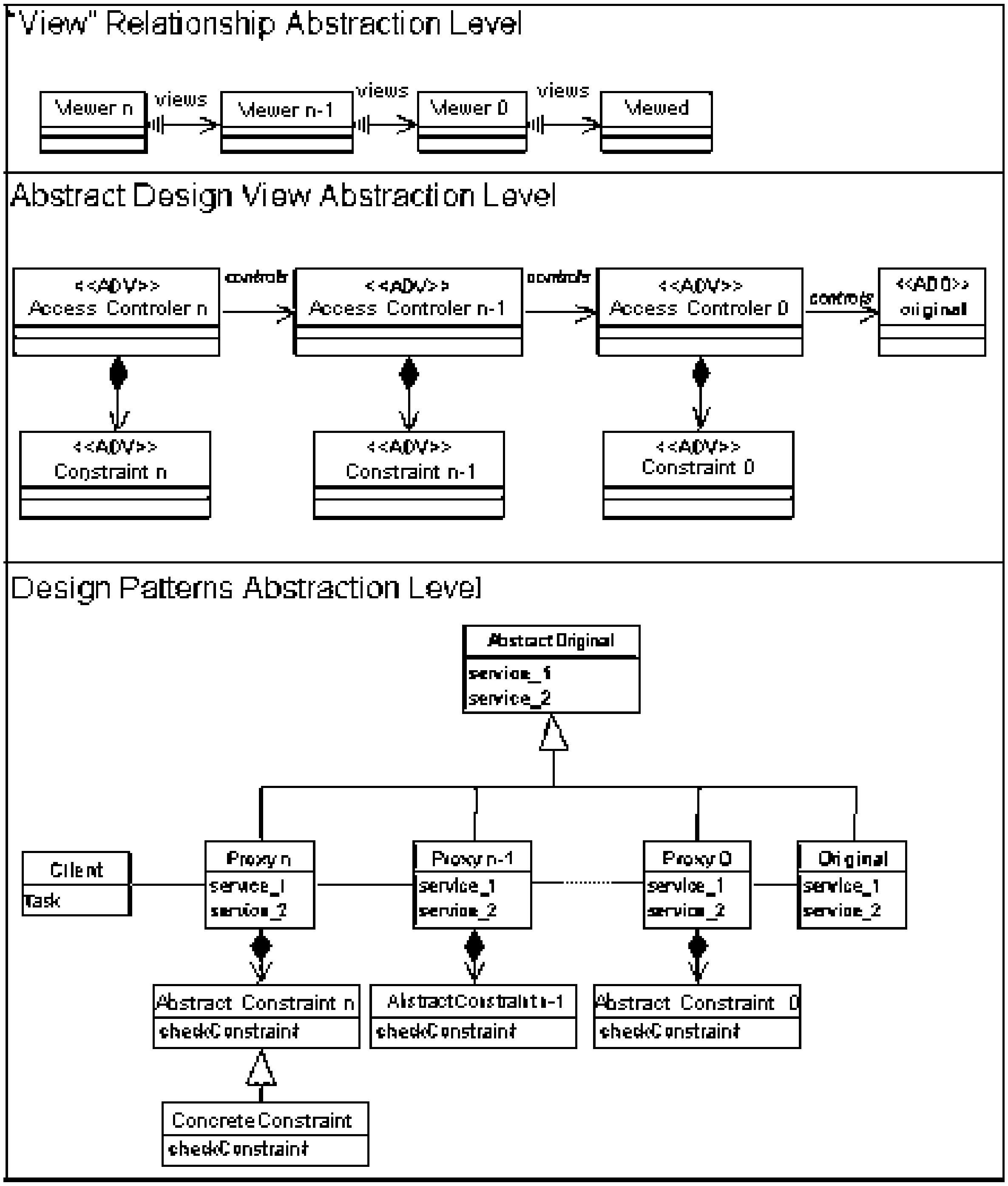

Figure 5: Realizing the "*Views*" relationship to design pattern

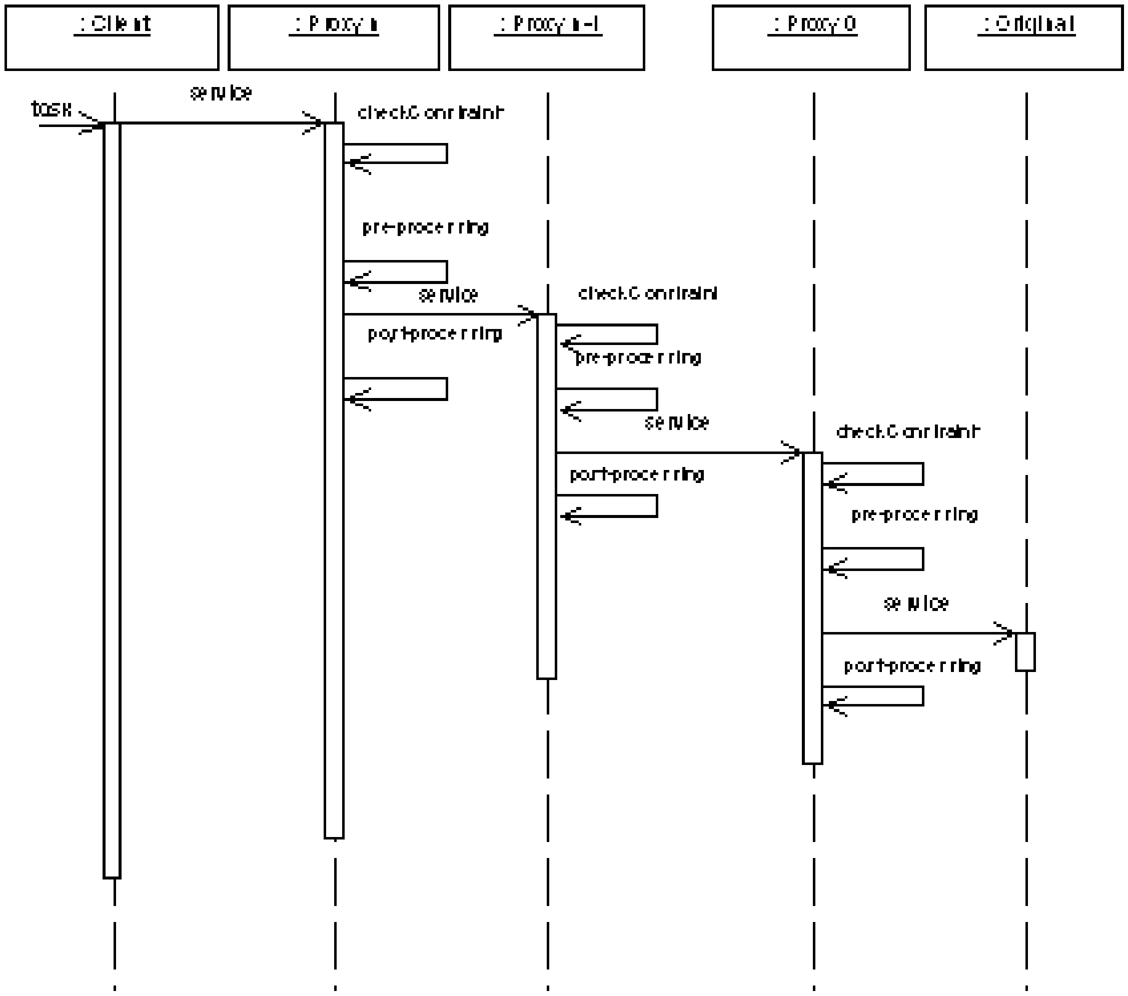

Figure 6: The UML Sequence diagram for the expanded Proxy Pattern

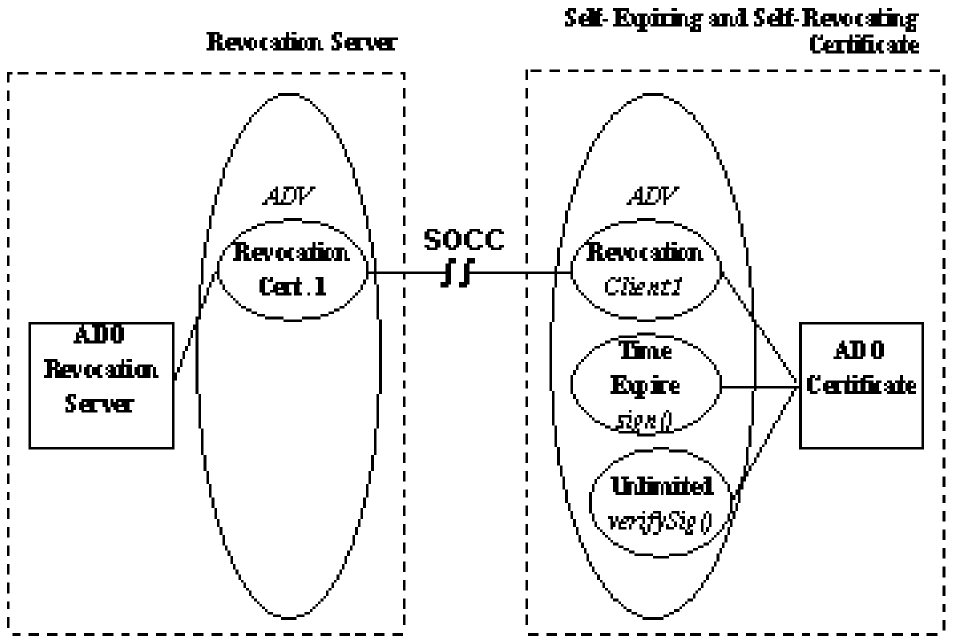

Figure 7: Self-Expiring Self-Revocating Certificate using SOCC's

in another design.

8 Example: A Self-Expiring Self-Revocating Certificate Architecture

In this section we will use the word certificate to mean an X.509-like Public Key certificate, as described in [28]. As a simplification, we are modeling this certificate as having only one time-constraint, a date limit on its use. However, this limit only applies to signing or encrypting new objects. This certificate should be able to verify if the signature of a signed object is authentic at all times. Therefore, we have a time-constraint and also an unconstrained use of the certificate.

We will consider that the method sign() does the signing of objects and verifySig() verifies signatures. Thus, in figure 7, the certificate is the ADO, protected by a composite ADV. This composite ADV has two ADVs: one that guards the sign() method and other the verifySig() method. The first enforces a time constraint, and the other provides no restriction on its use.

In figure 8, we have the certificate example modeled using an expanded

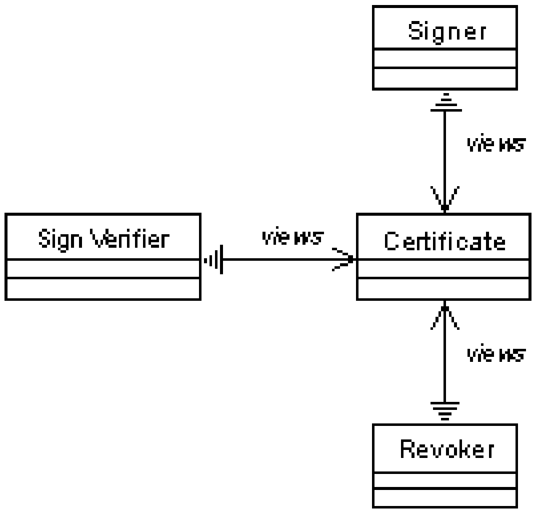

Figure 8: A "*Views*" model for the Certificate

UML that covers the "views" relationship. This model can be realized using the Hybrid Architecture, as shown in figure 9. UML provides the use of stereotype as one of the language extension mechanisms [3]. The ADVs are represented by the <<ADV>> stereotype and ADOs by the <<ADO>> stereotype. The ADO is modeled as the Certificate class, and the composite ADV as CertificateAccessController, with the TimeExpiringConstraint and UnlimitedConstraint as constraint ADVs. In this case, we have two subject views of the ADO: an object signer with time limit and an unlimited signature verifier.

After an architecture of ADVs was introduced, the proper mapping to design patterns is straightfoward, as in figure 9. Java RMI was used to model the SOCC objects.

One important issue is the complete separation of the application code from the constraint code, clearly visible in figure 9. We have a separate class hierarchy for the constraints and the application code.

Revocation is usually considered as part of the backend architecture. Every certificate has a unique identification. When it is revoked, this id is placed in a Certificate Revocation List (CRL) [28]. When the certificate is to be used, its identification is checked against the CRLs. If the certificate is revoked, the certificate is useless. Otherwise, the authentication process continues. A good discussion of CRLs and authentication architectures and solutions can be found in [27].

On the other hand, revocation can also be modeled as a client ADV connected to an ADV from the server, as in figure 7. When the certificate is

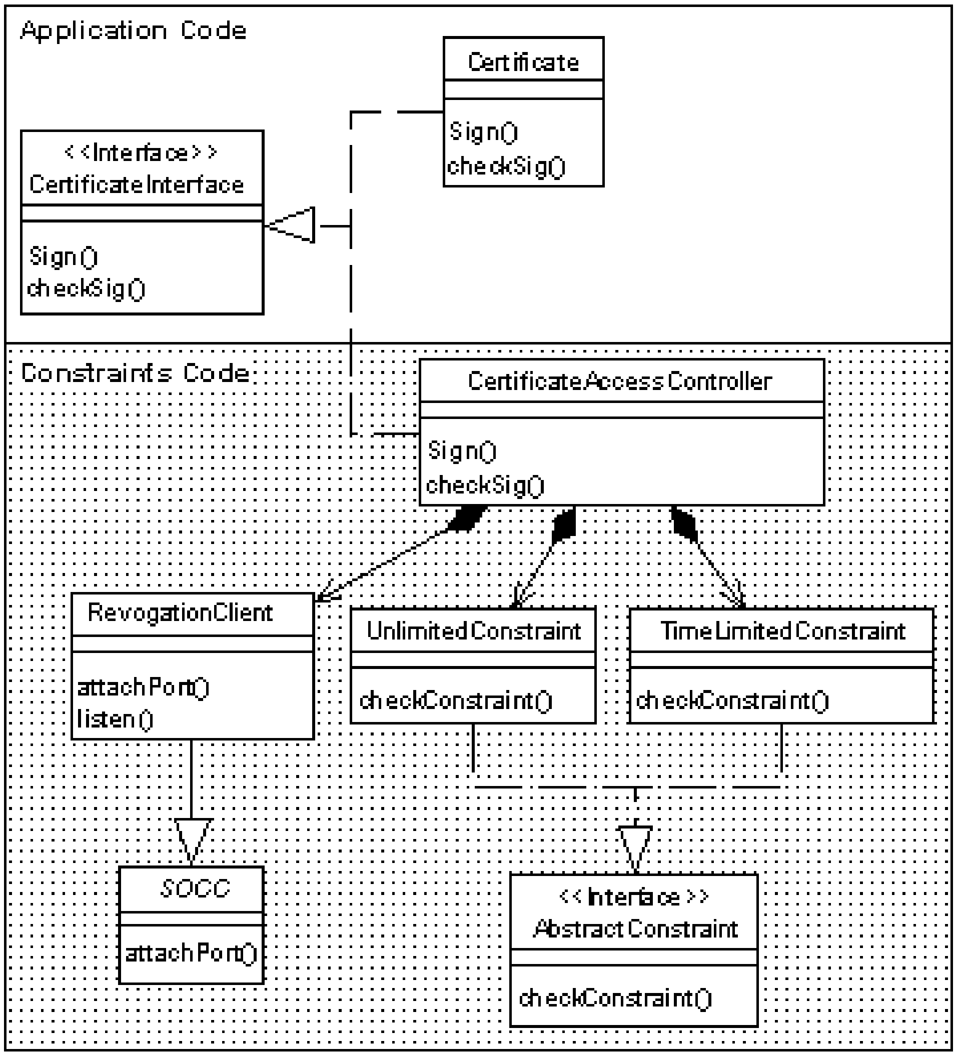

Figure 9: UML model for certificate model example

to be revoked, the server ADV warns the client ADV, that changes the client ADO state to invalid, rendering it useless. In this case, the certificate gains a self-revocation property. The security of the SOCC between the revocation server and the certificate ensures that the revocation process is protected from attacks of any kind. This model, however, depends on the capacity of the client to stay on-line, and is perhaps more suited for distributed systems where all parties remains on-line most or all the time. It is important to notice that this model is complementary to the CRL model, since the revocation server must have CRLs to revoke the client ADVs which are listening.

Figure 10 shows the source code in Java for the class CertificateAccess-Controller. The CertificateAccessController implements for each of the methods of its common interface with the Certificate object (the ADO), the polling to the constraint ADVs. As an example, the limit date for this certificate is October 25th, 1972, but any other date would do just as well. Notice that the ADO is a private property of the ADV. This way, no one but the ADV has references to the ADO, thuss3 preserving the ADO encapsulation.

In figure 11 we have the source code listing for the TimeExpiringConstraint. This ADV enforces a time limit constraint. In this example, the time limit is checked with the local time. In a real world application, this ADV would probably request the time from a secure time server, but this does not invalidate our example. If this class was the UnlimitedConstraint, the checkConstraint() method would return always true.

Figure 12 shows the source code for the RevocationClient class. This class extends the SOCC ADV class, and throws the CertificateRevoked exception. Whenever this exception is received, the certificate can be destroyed or the ADO can trigger its state to invalid once it catches the exception.

Therefore, this example shows that the certificate code is completely unaware of its use policy. Nonetheless, the use of separate hierarchies allows us to separate the code reuse and create generic time constraints that act as simple components for other applications.

9 Conclusion

In this paper a model for capabilities using Abstract Design Views (ADVs) has been proposed which is richer than the traditional capability model. Full separation of concerns and independence of the source code of the application from the access control is also achieved using ADVs. Thus, by using ADVs for modeling constraints, designers are able to reuse them in a new context.

A distributed access control model was also proposed with the use of Secure Object Communication Channels (SOCCs). This type of communi-

```
public class CertificateAccessController implements CertificateInterface
{
    // We have the Certificate ADO
    private static Certificate pCertificate;

    // There is the time expire constraint
    private static TimeExpiringConstraint pTimeExpiringConstraint;

    // There is the unlimited constraint
    private static UnlimitedConstraint pUnlimitedConstraint;

    // Revocation ADV listener
    private static RevocationClient pRevClientADV;

    CertificateAccessController() throws CertificateRevoked {
           Date pCurrDate = new Date( 72, 8, 25 );
           // any date will do, let's pick 8-25-72 as an example
           pTimeExpiringConstraint = new TimeExpiringConstraint( pCurrDate );
           // wait for revocation
           pRevClientADV.listen();
    }
    public boolean Sign()  {
           if ( pTimeExpiringConstraint.checkConstraint() == true ) {
                pCertificate.Sign();
                return( true );
           }
           else {
           // The Certificate has expired!!! Attention!
           return( false );
           }
    }
    public boolean checkSig() {
           if ( pUnlimitedConstraint.checkConstraint() == true ) {
                pCertificate.checkSig();
           }
           return( true );
    }
}
```

Figure 10: Source code for the CertificateAccessController class

cation channel benefits from encryption technologies, allowing secure connections at the object abstraction level.

Finally, the use of ADVs in access control modeling provides the user with a pure object-oriented and highly distributed model. Since the goal was achieved without the use of meta-objects, this model can be implemented in any object-oriented language, from C++ to Java.

Using the architectures shown in this paper, it was shown that security is possible at the application level. These allows reuse of constraints and a highly distributed approach, which is well suited for e-commerce applications requirements.

In a near future we intend to study the mapping of the "views" relationship to design patterns in a more general way. We are in the process of

```
public class TimeExpiringConstraint implements AbstractConstraint
{
    private static Date p_pDate;
    private static Calendar pCalendar = Calendar.getInstance();

    TimeExpiringConstraint( Date pDate ) {
           p_pDate = pDate;
    }
    public boolean checkConstraint() {
           Date pCurrDate = pCalendar.getTime();

           if ( p_pDate.before( pCurrDate ) == true ) {
              return( false );
           }
           return( true );
    }
}
```

[!hbp]

Figure 11: Source code for the TimeExpiringConstraint class

```
public class RevocationClient extends SOCC
{
    RevocationClient() {
    }
    public void listen()  throws CertificateRevoked  {
           // attach to a port
           attachPort();
    }
    public void attachPort() {
           // attaches this ADV to a port
           // creates the SOCC
    }
}
```

Figure 12: Source code for the RevocationClient class

developing a highly distributed Public-Key Infrastructure (PKI) [27] framework using SOCCs and constraint ADVs. Other uses for Abstract Design Views in security architectures are also being pursued. A UML extension for Abstract Design Views and Abstract Design Objects will also be developed.

10 Note to the reader

This work is part of an IBM Brazil project at the TecComm/LES project in PUC-Rio, Brazil.

Many of the technical reports mentioned in this paper are available via anonymous ftp from csg.uwaterloo.ca at the University of Waterloo. The names of the technical reports are in the file "pub/ADV/README" and electronic copies of the reports in postscript format are in the directories "pub/ADV/demo", "pub/ADV/theory", and "pub/ADV/theory".

References

[1] D. D. Cowan, C. J. P. Lucena, and P. S. C. Alencar, "Abstract Data Views as a Formal Approach to Subject-oriented Programming," technical report, Computer Science Department and Computer Systems Group, University of Waterloo, Waterloo, Ontario, Canada, May 1995.

[2] W. Harrison and H. Ossner, "Subject-oriented programming (a critique of pure objects)," in *OOPSLA '93*, 1993.

[3] OMG, *OMG Unified Modeling Language Specification Version 1.3*, June 1999.

[4] D. D. Cowan and C. J. P. Lucena, "Abstract data views: An interface specification concept to enhance design for reuse," *IEEE Transactions on Software Engeneering*, vol. 21, March 1995.

[5] M. Aksit, "Separation and composition of concerns," in *Proceedings of the ACM Workshop on Strategic Directions in Computer Research*, June 1996.

[6] G. Booch, I. Jacobson, and J. Rumbaugh, *The Unified Modeling Language User Guide.* Addison-Wesley, October 1998.

[7] P. S. C. Alencar, D. D. Cowan, and L. C. M. Nova, "A formal theory for the views relationship," tech report, Computer Science and Computer Science Department, University of Waterloo, 1998.

[8] L. C. M. Nova, “A formalization of an extended object model using views,” phd thesis, Computer Science Department and Computer Systems Group, University of Waterloo, Waterloo, Ontario, Canada, March 2000.

[9] D. D. Cowan, R. Ierusalimschy, C. J. P. Lucena, and T. M. Stepien, “Abstract Data Views,” *Structured Programming*, vol. 14, pp. 1–13, January 1993.

[10] D. D. Cowan, T. M. Stepien, R. Ierusalimschy, and C. J. P. Lucena, “Application integration: Constructing composite applications from interactive components,” *Software Practice and Experience*, vol. 23, pp. 255–275, March 1993.

[11] R. Helm, I. M. Holland, and D. Gangopadhyay, “Contracts: Specifying behavioral compositions in object-oriented systems,” in *OOPSLA'90*, pp. 169–180, 1990.

[12] D. D. Cowan, L. F. Barbosa, R. Ierusalimschy, C. J. P. Lucena, and S. B. de Oliveira, “Program design using abstract data views - an illustrative example,” Technical Report 92–54, Computer Science Department, University of Waterloo, Waterloo, Ontario, Canada, December 1992.

[13] C. J. P. Lucena, D. D. Cowan, and A. B. Potengy, “A programming model for user interface compositions,” in *Anais do V Simpósio Brasileiro de Computação Gráfica e Processamento de Imagens – SIBGRAPHI'92*, (Aguas de Lindoia, SP, Brazil), November 1992.

[14] A. B. Potengy, C. J. P. Lucena, and D. D. Cowan, “A programming approach for parallel rendering applications,” technical report, Computer Science Department and Computer Systems Group, Univ. Waterloo, Waterloo, Ontario, Canada, March 1993.

[15] Watcom Int. Corp., *WATCOM VX-REXX for OS/2 Programmer's Guide and Reference*, 1993.

[16] D. Smith, “Abstract data views: A case study evaluation,” Technical Report 94–19, Computer Science and Computer Science Department, University of Waterloo, Waterloo, Ontario, Canada, April 1994.

[17] D. D. Cowan, C. J. P. Lucena, and R. G. Veitch, “Toward caai: Computer assisted application integration,” Technical Report 93–17, Computer Sciece Department and Computer System Group, Univ. Waterloo, Waterloo, Ontario, Canada, January 1993.

[18] O. Nierstrsz, S. Gibbs, and D. Tsichritzis, "Component-oriented software development," *Communications of the ACM*, vol. 35, pp. 160–165, September 1992.

[19] G. Wiederhold, P. Wegner, and S. Ceri, "Toward megaprogramming," *Communications of the ACM*, vol. 35, November 1992.

[20] D. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, vol. 15, December 1972.

[21] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[22] L. M. F. Carneiro, M. H. Coffin, D. D. Cowan, and C. J. P. Lucena, "User interface high-order architectural models," Technical Report 93–14, Computer Science and Computer Science Department, University of Waterloo, Waterloo, Ontario, Canada, 1993.

[23] S. J. Mullender and A. S. Tanenbaum, "The design of a capability-based distributed operating system," *The Computer Journal*, vol. 29, pp. 289–300, March 1986.

[24] Sun Microsystems Computer Corporation, *The Java Platform*, May 1996.

[25] T. Riechmann and F. J. Hauck, "Meta objects for access control: Extending capability-based security," in *New Security Paradigms Workshop*, 1997.

[26] T. Qian and W. Liao, "Active capability: An application specific security and protection model," tech. rep., University of Illinois at Urbana-Champaign, Illinois, USA, January 1996.

[27] C. Adams and S. Lloyd, *Understanding Public-Key Infrastructure: Concepts, Standards, and Deployment Considerations.* Macmillan Technical Publishing, first ed., 1999.

[28] Int'l Telecommunications Union, Geneva, *Recommendation X.509 – Information Technology - Open Systems Interconnection - The Directory: Authentication Framework*, June 1997.

[29] B. Schneider, *Applied Cryptography: Protocol, Algorithms and Source Code in C.* Wiley, second ed., 1996.

[30] A. J. Menezes, P. C. Oorschot, and S. A. Vantone, *Handbook of Applied Cryptography.* CRC Press, October 1996.

[31] R. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public key cryptosystems," *Communications of the ACM*, February 1978.

[32] Federal Information Processing Standards, *Data Encryption Standard (DES), Federal Information Processing Standards Publication 46-2*, December 1993.

[33] W. Stallings, *Cryptography and Network Security: Principles and Practice.* Prentice Hall, second ed., 1998.

[34] T. Dierks and C. Allen, "The TLS Protocol Version 1.0, internet request for comments 2246," January 1999.

[35] Sun Microsystems Computer Corporation, *Java Secure Socket Extension (JSSE) 1.0.1 API Specification*, March 2000.

[36] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture: A System of Patterns.* Addison-Wesley, 1996.