Security Benefits from Software Architecture

C. Bidan and V. Issarny

IRISA / INRIA, Campus de Beaulieu, RENNES Cedex 35042, FRANCE

Abstract. In today's field of distributed software architectures there is a need for environments allowing the easy development of applications consisting of heterogeneous software modules and having various Quality of Service requirements (e.g., timeliness, availability or security). System customization using middleware-services is a promising solution to deal with the coexistence of multiple applications with different Quality of Service requirements.
From the security point of view, the goal for system customization is to permit the interoperation among applications having different, possibly inconsistent security constraints. This paper demonstrates how the software architecture paradigm is beneficial for addressing security issues in distributed systems through system customization. The software architecture paradigm allows the application developer to abstractly specify security-related requirements. Then, our framework takes in charge the system customization to meet these requirements. The practical use of our approach is also addressed by discussing its integration in a *configuration-based* distributed programming environment.

1 Introduction

Due to the progress in computer science, open distributed computing systems are now eligible for supporting a wide class of applications, having different Quality of Service (QoS) requirements (such as timeliness, availability or security). One promising approach to deal with the coexistence of multiple applications with different QoS requirements is system customization using middleware-services [4]. In order to hide the customization process from application developers, we need a framework to specify QoS requirements that takes in charge the system customization by selecting appropriate middleware-services [16]. The software architecture field provides an adequate basis to support such a framework.

The software architecture field has emerged in order to ease the development of complex applications (or software systems) by fostering software re-use, evolution, analysis, and management (e.g., [25, 28]). Although the software architecture field is continuously evolving, the clean separation between software components (e.g. computational units) and their interactions is widely accepted. In that framework, an *Architecture Description Language* (ADL) allows system specifications in terms of the three following abstractions [28]:

(i) *Components* that abstractly define computational units written in any programming language,
(ii) *Connectors* that abstractly define types of interactions (e.g., pipe, client-server) among components, and

(iii) Configuration that defines a system structure (i.e., a software architecture) in terms of the interconnection of components through connectors.

Connectors may be viewed as the *glue* that binds the components; they define the set of *roles* that the components must play to communicate, that is to say, the set of *properties* of, and *relationship* among, the components [28, 12]. Reasoning at the level of software architectures provides the adequate abstractions for system customization using middleware-services: QoS properties can be provided by a certain type of connectors in which roles include specifications of QoS-related connector behaviors (i.e., architectural style for connectors that is derived from the QoS requirements [25]). Work in the software architecture research field has been concentrating on the specifications of communication architectures, that is to say, the communication protocol used for handling component interactions (e.g. pipe, RPC, shared memory) [28, 1]. Whereas the referenced work focuses on application communication patterns, we are concerned with the QoS that is guaranteed when using a given connector. QoS-related customization can be achieved as follows. First, the application QoS requirements are defined within the specifications of connectors provided that each interaction is characterized by a unique connector. Dually, *base connectors* (i.e., connectors that are available within the environment) and *system-level software components* (i.e., middleware-services) declare the QoS properties they provide. Then, given the formal specifications of QoS properties, software specification matching [32] is used to retrieve a set of system-level software components together with (base) connectors, so that the conjunction of their QoS properties matches the application requirements [14]. The connector built through the interconnections of selected system-level components and (base) connectors is called a *customized connector* in the following.

Interactions among components, possibly having different QoS requirements, are not taken into account in the above customization model, for which a unique connector is declared for each interaction. However, in open distributed systems interactions among entities may occur dynamically. During the interaction between two components, we must consider requirements of both components (i.e., each component can specify their own connectors). By composing these requirements, we are able to infer the requirements for the interaction according to both components. These latter requirements can be used to build the customized connector. From the security point of view, the complex interactions among components stretch the security problems in open distributed systems [24, 8], and require coping with the coexistence of multiple security constraints.

In this paper, we demonstrate how the software architecture paradigm is helpful for addressing security of open distributed systems: at the compilation time based on the declaration of software components having security requirements (e.g., by *security managers*), a customized connector is built to meet both requirements. After introducing security issues for open distributed systems in section 2, we apply in section 3, system customization to security properties. This allows us to provide a framework enabling *security managers* to implement distributed systems which support multiple security constraints. We then present implementation design in the

ASTER configuration-based environment in section 4. Finally, we conclude in section 5 by addressing related work and referring to future work.

2 Security issues for open distributed systems

Security means protection against unauthorized attempts to access information [11, 17, 31]. It is concerned with *confidentiality* (information is disclosed only to users authorized to access it), *integrity* (information is modified only by users who have the right to do so, and only in authorized ways) and *availability* (use of the system cannot be maliciously denied to authorized access). In the remainder, we focus only on the first two constituents of security, the issue of availability being beyond the scope of this paper as it also relates to fault-tolerance issues[1].

Security (i.e., data confidentiality and integrity) is an important issue in any distributed system. Security is enforced using *security functionalities* such as *encryption*, *authentication* and *access control*.

Data encryption consists of making the information either illegible (encryption in order to ensure data confidentiality) or unalterable (digital signature in order to ensure data integrity), or both. Encryption is used for protecting stored and exchanged information (e.g., *via* communication network) against reading or modification.

Authentication permits to ensure the entity's identity, that is to say to verify that the entity (e.g., a user or a process) is actually the one that it claims to be. More specifically, the *authentication protocol* enables to associate each system operation to a unique real user, allowing the checking of whether the operation is authorized or not. An example of *authentication protocol* is given by the UNIX system: it requires the user to enter his name, followed by his password. Let us remark that authentication may be considered as the fundamental security mechanism: if it is not provided, an entity is able to claim to be any entity, and thus access any information.

Access control governs operations on system's entities. For instance, in a file system, access control consists of checking whether users are allowed to access files. More generally, in a large distributed system, access control checks the interactions among entities. By extension, access control also deals with the information flow among entities. Let us remark that access control clearly depends upon authentication, the entity's identity having to be unique and unforgeable.

In open distributed systems, applications which have different security constraints may coexist. This coexistence results from rich and complex interactions among software components [24], and has led to the need for reasoning about different security constraints: a sound basis for describing the software architecture with respect to the security constraints is needed. This is the scope of this paper in which we use the software architecture paradigm to specify the security requirements, and to provide a system customization approach to build systematically connectors that meet these requirements.

[1] Let us notice that this "simplification" is often made in the computer security community through the following phrase: "I don't care if it works, as long as it is secure" [11].

In the above presentation, we have simplified some issues so as to not needlessly complicate this paper. In particular, we have not distinguished between the identification process (associating a user ID with a program) and the authentication process (associating the real user with the user ID), and we have not introduced the delegation process enabling entities (e.g., users) to delegate access privileges to other entities (e.g., program) [18].

To implement security functionalities, some security mechanisms are needed. One fundamental issue of security is proving the correctness of the security mechanisms according to security properties (i.e., the correctness of the system-level components used to implement security functionality) [2]. In the remainder, we assume the existence of a *Trusted Computing Base* (or TCB [9]) which contains "correct" security mechanisms that can be used safely to implement security functionalities.

3 Customized connectors for dealing with security

In traditional distributed systems, an application developer who wants to implement security functionalities has to implicitly manage the corresponding security mechanisms in the application source code (e.g., to implicitly call the encryption and decryption functions). This approach is detrimental to both flexibility and reusability. Besides, the management of interconnected systems is complex indeed impossible. In confined distributed systems, this approach may be considered as sufficient, a unique *security manager* having to manage security issues. On the other hand, the above approach is not suited for open distributed systems. In particular, such a system must support interactions among entities belonging to different *security domains*[2].

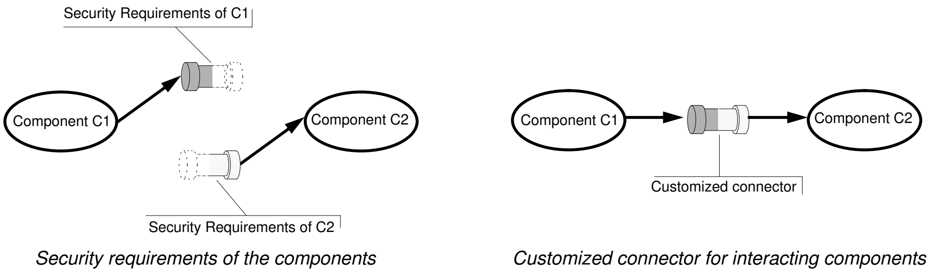

Fig. 1. Security requirements and customized connectors

Because security issues (in term of encryption, authentication and access control) relate to interactions among software components, it is natural to associate security properties with connectors. We extend the connector specifications so as to include the (application developer) security requirements for the interactions among given software components. At the time of the interaction between any two software components, either a unique connector σ is specified for this interaction, or both

[2] The *security domain* of a given *security manager* is the set of all the system entities he manages [11].

software components specify their own connectors σ_1 and σ_2 respectively (i.e., their own security requirements). In the first case, a customized connector is built to meet the requirements specified in the connector σ. In the latter case, we compose the requirements of both connectors σ_1 and σ_2 in order to specify the connector $\sigma_{1,2}$ for the interaction of the software components. The connector $\sigma_{1,2}$ allows one to build the customized connector that meets the security requirements (see figure 1). As discussed in the introduction, the customization process consists of selecting a set of system-level software components that provide security functionalities together with (base) connectors. Note that the system-level components and the (base) connectors have to belong to the TCB.

In order to automatically build customized connectors that meet the security requirements, we have: *i)* to specify security properties for encryption, authentication and access control (i.e., to describe security requirements), and *ii)* to reason about these specifications so as to compose and compare them (i.e., for composing security requirements and for selecting the system-level components).

In the following subsections, we introduce specifications of encryption, authentication and access control which allow us to describe security requirements. We also address comparison and composition of security specifications.

3.1 Encryption specifications

Data is qualified as *plaintext* or *cleartext* when access to this data allows access to the information which is contained. The process of disguising data in such a way as to hide the contained information is *encryption*, and the resulting data is qualified as *ciphertext*. The process of turning ciphertext back into plaintext is *decryption*. An *encryption algorithm* is then composed of an *encryption function* and the corresponding *decryption function*. The encryption function may be used to ensure either confidentiality or integrity.

In general, the encryption and decryption functions are not secret: the security of the encryption algorithm is based on the use of keys. The encryption function takes as input the plaintext and the *encryption key* to compute the ciphertext. In the reverse process, given the ciphertext, the plaintext is computed by using the corresponding *decryption key* and the decryption function (see figure 2)[3].

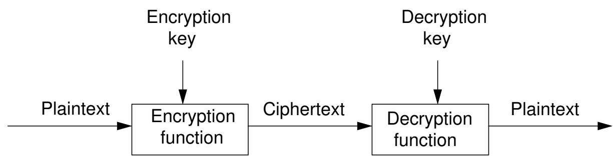

Fig. 2. Encryption and decryption with keys

[3] For more details about cryptography, see [27].

Specifying encryption requirements Specifying security properties relating to encryption relies on the specification of the encryption algorithm that is used (e.g., by its name: RSA, DES, ...), as well as on the specifications of parameters describing the algorithm behaviors. For instance, parameters may include:

- the algorithm usage, that is to say, if the encryption algorithm is used to cipher the data (for ensuring confidentiality), to sign the data (for ensuring digital signature) or to hash the data (for detecting manipulation);
- the key management, that is to say, the key size (e.g., 512 bits for the RSA algorithm, or 64 bits for the DES algorithm), the support used to find and/or to store the keys (e.g., user's brain, ROM chip or smartcard), the key generation algorithm used to generate secret temporal keys (e.g., random-bit strings algorithms), the lifetime of the secret keys, and so on.

Thus, encryption specification allows one to describe the requirements for encryption as a parameterized encryption algorithm.

As an illustration, an application developer may specify the following requirements for encryption:

Security
  Cipher Algorithm : DES
    Temporal Key : RC4
    Key Size : 64
    Storage : ROM
    Lifetime : 1 hour
  Signature Algorithm : RSA
    Key Size : 512
    Storage : Smartcard

where boldface is used to denote specification keywords. In the above specification, the developer wishes to use the cipher algorithm DES, with 64 bits' temporal keys which are generated for 1 hour by the RC4 algorithm, and stored in ROM chip. He requires also the use of the RSA signature algorithm, the 512 bits' keys being stored in smartcard.

In order to decipher data, we must use the decryption function and key, corresponding to the encryption function and key that were used to cipher the original data. During the interaction of two software components, each of them can have encryption requirements. In this context, composing and comparing these requirements are needed so as to ensure that both components use dual encryption algorithms (i.e., to build customized connectors according to encryption requirements of both components).

Composing encryption requirements Due to the symmetry of the encryption and decryption functions, we can not compose and compare two different encryption algorithms (e.g., DES and RSA). However, the composition of encryption requirements can be achieved as follows. We enable the application developer to specify a list of encryption algorithms within connectors. We index each encryption algorithm with the trust degree that the application developer assigns to them [4]. Given two connectors with encryption requirements, the encryption algorithms can be ordered according to the above trust degrees. Then, the encryption algorithm of the connector corresponds to the encryption algorithm (or one of the encryption algorithms) that belongs to both sub-connectors and has the highest trust degree.

Once the encryption algorithm is selected, we are able to reason about its parameters, the connectors being able to specify different values. Like the ordering of encryption algorithms, we define an order on the parameters of encryption algorithms according to the same approach. For instance, let us consider the key size[5]. The application developer may consider that, given an encryption algorithm, the use of 1024 bits' keys instead of 512 bits' keys is acceptable according to the trust degree of the resulting encryption functionality. More generally, the lattice resulting from such an order allows one to compare and compose encryption requirements with respect to the parameter values of the selected encryption algorithm, through specification matching [32].

Given two connectors with encryption requirements (i.e., lists of parameterized encryption algorithms), the (parameterized) encryption algorithm resulting from their composition is the greatest parameterized encryption algorithm *Encrypt* with respect to the trust degree that each application developer (independently) assigns to the encryption algorithms and to their parameter's values. Then, the customization process can be achieved if and only if we can build a customized connector that meets the parameterized encryption algorithm *Encrypt* (i.e., the encryption algorithm of the customized connector is greater than the parameterized encryption algorithm *Encrypt*).

3.2 Authentication specifications

Authentication of entities (e.g., user, process, software component, system) allows to verify that the entities are who they claim to be. An *authentication protocol* specifies the authentication process [6], that is to say, the entities which participate, the exchanged messages among these entities and the format of these messages (i.e., plaintext or ciphertext). Although the main goal of authentication protocols is the authentication of entities, some of them have evolved to deal with additional security properties, such as mutual authentication or temporal key distribution. Authentication protocols needing encryption algorithms for ciphering and/or signing the exchanged messages, another evolutions have consisted of implementing encryption algorithms (for instance, RSA instead of DES). We characterize the former evolution as security semantic evolutions, and the latter evolution as security implementation evolutions.

Let us remark that the authentication protocol resulting from semantic and/or security evolutions may be viewed as an extension of the original protocol, and hence,

[4] In the context of software architectures, the notion of trust degree is close to the notion of weighted properties of the architectural form [25].

[5] An order can also be introduced for any parameter including the lifetime property, the temporal key property (by ordering the generator algorithm according to the trust degree given by the application developer).

can be used to implement the original protocol (like the compatibility among the versions of software components). For instance, let us consider an authentication protocol with simple authentication, and suppose that an extended version of this protocol allows to ensure mutual authentication. Then, the latter version can be used to provide simple authentication.

Specifying authentication requirements Authentication specification relies on the specification of the (original) authentication protocol (e.g., Needham-Schroeder protocol, Kerberos protocols, CCITT X.509 protocol [6]). We further parameterize these specifications according to their semantic evolutions (i.e., simple authentication, mutual authentication, key distribution) and their security implementations evolutions (i.e., implementation version, type of encryption algorithms used to cipher and to sign).

Finally, authentication specification also has to include encryption specification in order to describe the encryption algorithm (see §3.1) used to ensure confidentiality and integrity of the exchanged information. Notice that an authentication protocol may use random generator in order to generate temporal keys or to generate timestamps.

As an illustration, authentication specification for a Needham-Schroeder authentication protocol that uses RSA encryption algorithm for both ciphering and signing exchanged message follows:

Security
  Authentication Protocol : Needham-Schroeder
    Semantic :
      Authenticate : **Mutual**
      Version : *number of the version*
    Encryption
      Cipher Algorithm : RSA
        Key Size : 512bits
        Storage : ROM
      Signature Algorithm : RSA
        Key Size : 512bits
        Storage : ROM

Implementing authentication between two entities requires the use of at least the same authentication protocol. During the interaction of two software components, each of them can have authentication requirements. In this context, composing and comparing these requirements are needed so as to ensure that both components use dual authentication algorithms.

Composing authentication requirements As in the encryption specifications, by enabling application developers to specify an ordered list of authentication protocols within connectors, we are able to compose connectors having different authentication requirements (i.e., different lists of authentication protocols) according to the trust degree that each application developer assigns to each protocol. Given two connectors with authentication requirements, the authentication protocol of the resulting connector corresponds to the authentication protocol that belongs to both of them and has the higher trust degree.

Once the authentication protocol is selected, we are able to reason about its semantic and security evolutions. We order the evolutions according to the trust degree that each application developer assigns to the parameter's values. Then, the lattice resulting from such an order allows us to compare and compose authentication requirements with respect to the parameter values of the selected authentication protocol, through specification matching [32].

Given two connectors with authentication requirements (i.e., lists of parameterized authentication protocols), the (parameterized) authentication protocol resulting from their composition is the greatest parameterized authentication protocol *Authen* with respect to the trust degree that each application developer (independently) assigns to the authentication protocols and to their parameter's values. Then, the customization process can be achieved if and only if we can build a customized connector that meets the parameterized authentication protocol *Authen* (i.e., the authentication protocol of the customized connector is greater than the parameterized encryption algorithm *Authen*).

To summarize, implementing encryption and/or authentication between two software components require to use at least dual algorithms and/or protocol. As a consequence, the composition of security-related requirements amounts to a selection in ordered lists of algorithms and/or protocols specified within connectors (i.e., specification matching with respect to the trust degree that each application developer assigns to them). In addition, ordering the encryption algorithms and authentication protocols according to their parameter values allows also to use specification matching to compare and compose encryption and authentication requirements. With this approach, customized connectors can be built to meet component requirements. According to the connectors with encryption and/or authentication requirements, we are able to select a set of system-level components (which belong to the TCB) and a set of (base) connectors (also belonging to the TCB), in order to construct the customized connectors for the required properties.

3.3 Access control specifications

An *access control policy* (or *ACP*) defines the set of rules, called *access control rules* (or *ACR*), that specify for a pair of entities (ϵ_1, ϵ_2) (e.g. users, processes, files, software components, etc) whether ϵ_1 is authorized to access ϵ_2 or not.

The interaction among entities which do not belong to the same security domain relies on two access control policies, each entity being able to have its own access control constraints. Unlike the encryption and authentication functionalities, the access control does not need symmetric implementation.

Deciding for granting accesses between two entities that belong to different security domains raises the issue of composing the corresponding access control policies: the resulting access control policy (called *composed access control policy* in the following) will allow to check whether the interaction between the entities is authorized.

However, various policies can result from the composition of two access control policies, as exemplified below.

Examples Let us consider a file system where we have two sets of users A and C, and two sets of files B and D. We define two access control policies $\mathcal{P}_F^1$ and $\mathcal{P}_F^2$: $\mathcal{P}_F^1$ authorizes users in A to access files in B, and $\mathcal{P}_F^2$ authorizes users in C to access files in D.

First, we compose these sub-policies so as to build the following composed policy, called $\mathcal{P}_F$: the users belonging to both sets A and C (denoted as $A \cap C$) are authorized to access files in $B \cap D$, users in A but not in C (denoted as $A - C$) are authorized to access files in $B - D$, and finally, users in $C - A$ are authorized to access files in $D - B$. $\mathcal{P}_F$ is a *restriction* of the participating policies: some accesses which are authorized in the sub-policies are denied by the $\mathcal{P}_F$ policy.

Let us now compose the sub-policies $\mathcal{P}_F^1$ and $\mathcal{P}_F^2$ so as to obtain the following policy, called $\mathcal{P}'_F$: the users belonging to $A \cap C$ are authorized to access files in B or in D (denoted as $B \cup D$), users in $A - C$ are authorized to access files in $B - D$, and finally, users in $C - A$ are authorized to access files in $D - B$. $\mathcal{P}'_F$ is consistent with both sub-policies: a user is authorized to access a file in $\mathcal{P}'_F$ if and only if the user is authorized to access this file in both $\mathcal{P}_F^1$ and $\mathcal{P}_F^2$.

Another composition of the sub-policies $\mathcal{P}_F^1$ and $\mathcal{P}_F^2$ could result in the following policy $\mathcal{P}''_F$: the users belonging to $A \cup C$ are authorized to access files in $B \cup D$. $\mathcal{P}''_F$ is an extension of the participating sub-policies (i.e.. A's users can access D's files).

Note that all the above composed policies may be regarded as secure in the context in which they are specified. Thus, there is a need for a framework allowing security managers to compose security policies in a controlled and secure way.

To deal with access control composition, our approach relies on the specifications of access control policies and on the definition of composition operators between these specifications.

Specifying access control policies Since in open distributed systems, it is not possible to specify an ACP that takes into account the whole set of the system's entities, we define an ACP not solely in terms of its *access control rules* but also in terms of its *access control domain*, that sets the system's entities to which the policy applies.

The access control domain of a policy is defined in terms of security classifications (i.e., security properties like *secret* entities, *confidential* entities, *unclassified* entities). An access control rule is a set of pairs of system entities verifying an *access control predicate*, i.e. a security property bearing on the system's entities.

So as to illustrate ACP specification, we consider the previous file system example where we omit reflexive accesses. The $\mathcal{P}_F^1$ ACP is defined by:

Security
  Access
    Domain
      Classif Cl-A : Users **with** *PropertyA*
      Classif Cl-B : Files **with** *PropertyB*
    Authorized
      (Entity **in** Cl-A) **to** (Entity **in** Cl-B)

that is to say, the security domain of the policy $\mathcal{P}_F^1$ consists of the users' classification A and the files' classification C. The policy $\mathcal{P}_F^1$ has a single access control rule that authorizes A entities to access B entities.

In our approach, a composed access control policy is built by composing two sub-policies. The domain and rules of the composed policy are defined in terms of the composition of its sub-policies' domains and rules.

Composing access control policies Given two classifications, we define three types of composition: their union, intersection and product. The union of two classifications allows one to extend them. On the other hand, the classification obtained by the intersection is a restriction of the sub-classifications. The product of two classifications permits one to distinguish the entities belonging to a single sub-classification from the entities belonging to both sub-classifications.

The definition of the composition of access control domains follows directly: it amounts to specify the type of composition for each pair of classifications of the sub-domains. Notice that the composed domain can then be composed with another access control domain.

Given two access control rules, we define two composition operators: the *logical or* and the *logical and* between their access control predicates. The *logical or* operator between predicates allows one to compose two access control rules in such a way that the resulting access control rule preserves all the accesses of the sub-rules. On the other hand, the access control rule resulting from the *logical and* operator authorizes access if and only if both sub-rules authorize the access.

Coupling the operators for composing classifications and the ones for composing access control predicates allows one either to restrict or to extend the access control rules of the sub-policies. In particular, using the intersection operator and the *logical and* operator for composing access control predicates, the access control rule resulting from the composition of two sub-rules verifies that the access is authorized if and only if it is authorized by both sub-rules.

Finally, in order to compose two access control policies, we have to specify all the composition operators between their classifications and between their access control rules. Introducing default operators for composing classifications and rules in the access control specifications allows to express the requirements of each connector concerning the operators to be used to compose its ACP with another one. Thus, the composition of two ACPs can be automatically computed: given two connectors with access control requirements, we are able to compose these requirements in order to specify the access control requirements of the composed connector.

We now illustrate the use of composition operators with the file system example. The composed policy $\mathcal{P}_F$ authorizes accesses between $(A-C)$ and $(B-D)$, between $(A \cap C)$ and $(B \cap D)$, and between $(C-A)$ and $(D-B)$.

The specifications of the $\mathcal{P}_F^1$ and $\mathcal{P}_F^2$ policies can be composed to give $\mathcal{P}_F$. The distinction among the classifications $(A-C)$, $(A \cap C)$ and $(C-A)$ (resp. $(B-D)$, $(B \cap D)$ and $(D-B)$) is obtained by the product operator of the classification A and C (resp. B and D). Finally, the use of the *logical and* operator between their access control rules enables us to generate the $\mathcal{P}_F$ access control rules.

We stop at this point in our discussion of ACP specification. In particular, we do not introduce formal denotation that allows to prove the completeness and the soundness of the access control specifications. The interested reader can refer to the companion paper [5] which define formally access control specifications and the aforementioned composition operators, as well as the notion of secure access control policy with respect to the proposed specifications.

Comparing access control specifications The above composition approach provides a basis for building the ACP resulting from the interactions among entities that belong to different security domains. In the context of software system, this approach allows us to compose connectors having different ACP specifications according to the composition operators that each application developer (e.g., *security manager*) has specified.

In order to build the (customized) connector that meets ACP requirements (i.e., the (composed) connector that describes the ACP constraints), we have to compare access control specifications. With respect to relationship between set of entities, the classifications (and then the security domains) can be ordered. Given two classifications Cl_1 and Cl_2, we introduce the *plug-in match* and *exact match* operators defined by:

- under *plug-in match*, the classification Cl_1 matches Cl_2 if and only if $Cl_1 \supseteq Cl_2$, that is to say, any Cl_2's entity belong to Cl_1, and
- under *exact match*, the classification Cl_1 matches Cl_2 if and only if $Cl_1 \equiv Cl_2$.

These operators enable us to compare connectors according to their access control domain specifications. On the other hand, the relation of logical implication provides a basis for comparing access control rules. Given two access control rules λ_1 and λ_2, we define the *plug-in match* and *exact match* notion between accesses:

- under *plug-in match*, the rule λ_1 matches λ_2 if and only if any access which is authorized (resp. denied) by λ_1 is also authorized (resp. denied) by λ_2, and
- under *exact match*, the rule λ_1 matches λ_2 if and only if λ_1 matches λ_2 under *plug-in match* and λ_2 matches λ_1 under *plug-in match*.

Thanks to the aforementioned *plug-in match* and *exact match* operators between classifications and access control rules, we are able to compare access control specification. Then, given the (composed) connector, we can select a set of system-level components (which belong to the TCB) and (base) connectors (also belonging to the TCB), in order to automatically construct the customized connector that meets the access control requirements.

4 Integrating security in Aster

The preceding discussion has allowed to demonstrate the adequacy of a software architecture approach for specifying and reasoning about security in open distributed systems. Specification matching enables us to build the customized connector that meets security requirements. To gain a practical benefit of this, application developers need a programming environment which allows software components to describe their security requirements in their interfaces. Aster is such a distributed programming environment (see [14]).

The Aster environment is a *configuration-based* environment (e.g., [26, 20]). It consists of: *i)* a *Module Interconnection Language* (Mil) [26, 10] which allows the application developers to describe the interfaces of its software components (i.e., the *components* according to Adl's terminology), and the interactions among them (i.e., the *configuration* according to Adl's terminology), and *ii)* the run-time system (i.e., an instantiation of a connector) which is possibly customized to satisfy the needs of a given application (e.g., security requirements) [14]. The Aster Mil defines the **requires** and **provides** clauses for declaring software component needs and provided properties respectively (i.e., connectors).

The Aster logical tool is responsible to select the system-level components with the corresponding connectors, that allow to build the *customized run-time system* (i.e., the customized connector) that meets QoS requirements [16].

Example. Figure 3 contains an example of interface declarations of a distributed file system, called FS, using the Aster Mil. First, the client's interface is declared. The keyword **client** denotes that the specific module issues requests for the declared operation. The declaration of the server's interface follows. The third block contains the construction of the file system application based on the declared interfaces, and is divided in two regions: *i)* the modules that participate, and *ii)* the bindings between requests and services.

We extend the Mil in order to integrate specifications of security constraints (*via* the **Security** clause) within the declaration of software components. For instance, figure 4 gives the interface of the file system, including the specifications of the $\mathcal{P}_F^1$ access control policy; we also specify that mutual authentication of clt-FS and srv-FS components *via* the Needham-Schroeder protocol is needed. Finally, thanks to the operators for composing and comparing security specifications, the logical tool can be directly extended to reason about security requirements.

```
interface clt-FS {
   client typeFD read(typeFD fd, typeBUF buf)
   client typeINT write(typeFD fd, typeBUF buf)
}

interface srv-FS {
   typeFD read(typeFD fd, typeBUF buf)
   typeINT write(typeFD fd, typeBUF buf)
}

hierarchical File-System {
   constituents clt-FS; srv-FS;
   bind
      (clt-FS)read : (srv-FS)read;
      (clt-FS)write : (srv-FS)write;
}
```

Fig. 3. Declaring FS's software components.

```
hierarchical FS-A_users-B_files {
   constituents clt-FS; srv-FS;
   bind
      (clt-FS)read : (srv-FS)read;
      (clt-FS)write : (srv-FS)write;
   requires
      Security
         Access
            Domain
               Classif Cl-A : clt-FS with PropertyA
               Classif Cl-B : srv-FS with PropertyB
            Authorized
               (Entity in Cl-A) to (Entity in Cl-B)
         Authentication Protocol : Needham-Schroeder
            Entities : clt-FS and srv-FS
            Semantic :
               Authenticate : Mutual
            Encryption
               Cipher Algorithm : RSA
                  Key Size : 512bits
                  Storage : ROM
               Signature Algorithm : RSA
                  Key Size : 512bits
                  Storage : ROM
}
```

Fig. 4. Declaring FS's security requirements.

5 Conclusion

Based on the software architecture paradigm, we have described an approach to specify and compose security requirements of software systems. The integration of this approach in the ASTER *configuration-based* environment provides a framework enabling the application developers (e.g., *security managers*) to specify security requirements of their software components. Thanks to its logical tool, the ASTER environment handles security specifications to build the customized runtime-system that meets these requirements.

The separation between security constraints and implementation issue is helpful not only for *security managers* who have to implement complex applications in a controlled and secure way, but also for *end-users* applications which can specify their own security requirements for interaction among their software components. The resulting runtime-system is then customized to meet the application requirements with respect to the system requirements (specified by *security managers*).

Related Work

The integration of security in large scale distributed systems is not a new concern [22]. The emerging object-based architectures [23, 7] renew the interest in security issues, and lead to the dissociation between implementation and security constraints [24, 8]. However, existing work does not deal with the composition of security constraints, which must be addressed in order to allow the interoperation of software components. On the other hand, recent work uses the object paradigm, and especially the abstract data type so as to ease the security implementation [30], but this work does not propose an automatic customization of the run-time system according to the security requirements.

Various models have been proposed in order to reason about security properties (e.g., see [19] for access control specifications, and [6] for authentication specifications). These models are introduced to check whether the security functionality verifies given security properties. However, these models do not deal with the composition of properties.

Concerning the composition of security properties, and more specifically, composition of access control policies, McLean [21] seems to be the first to propose a formal approach including composition operators: he introduces an algebra of security which enables him to reason about the problem of policy conflict, but he does not resolve the conflict (i.e., he does not allow to compose inconsistent access control policies). More generally, Hosmer [13] has introduced the notion of metapolicies, or "policies about policies", an informal framework for composing *security policies*. Bell formalizes this notion by considering the composition of two access control policies as a function (*policy combiner*) [3]. Our approach to compose access control policies is a practical alternative to Bell's proposal: we clearly identify the set of operators enabling security managers to compose access control policy in a controlled and secure way.

For the past decade, there has been considerable research and development in the area of using formal specification for determining whether two components match. However, to our knowledge, software specification matching has not been used to retrieve software components according to their security properties.

Finally, to our knowledge, using software architecture paradigm to reason about security properties and the integration of security specification in a configuration-based environment is not addressed elsewhere.

Future Work

We are currently working on the implementation of our approach by extending the Aster logical tool to automatically customized the runtime-system according to the security requirements. We are focusing more specifically on integrating the previous models and composition operators on access control policies. We are also examining the practical use of the Aster system for integrating security in concrete applications. These include the integration of security issues in a federated distributed file system [15].

Finally, given the treatment of security requirements, one application area for our results is the World-Wide Web (Www). We are currently examining the automatic customization of a base Www platform for ensuring information exchanges in a secure way. We also consider the use of our approach for reasoning about Java-applets, in order to customize the Java Virtual Machine [29] in a controlled and secure way.

References

1. R. Allen and D. Garlan. Formalizing architectural connection. In *Proceedings of the Sixteenth International Conference on Software Engineering*, 1994.
2. J. P. Banâtre, C. Bryce, and D. LeMétayer. Mechanical Proof of Security Properties. In *European Symposium on Research in Computer Security*, November 1994.
3. D. E. Bell. Modeling the Multipolicy Machine. In *Proceedings of the New Security Paradigm Workshop*, pages 2–9, August 1994.
4. P.A. Bernstein. Middleware: a Model for Distributed System Services. *Communication of the ACM*, 39(2), February 1996.
5. C. Bidan and V. Issarny. Dealing with Multi-Policy Security in Large Open Distributed Systems. Submitted for publication, May 1997.
6. M. Burrows, M. Abadi, and R. Needham. A Logic of Authentication. Technical Report 39, Digital Systems Research Center, February 1989.
7. M. Chapman and S. Montesi. Overall Concepts and Principles of TINA. Technical Report TB_MDC.018_1.0_94, TINA-C Document, 1995.
8. R.H. Deng, S.K. Bhonsle, W. Wang, and A.A. Lazar. Integrating Security in CORBA Based Object Architectures. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 50–61, May 1995.
9. Department of Defense Standard. Trusted computer system evaluation criteria. Technical Report DoD 5200.28-STD, December 1985.
10. F. DeRemer and H. Kron. Programming-in-the-Large versus Programming-in-the-Small. *IEEE Transactions on Software Engineering*, 2(2):80–86, June 1976.

11. M. Gasser. *Building a secure computer system.* Number ISBN 0-442-23022-2. Van Nostrand Reinhold, 1988.
12. D. Gelernter and N. Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):97–107, 1992.
13. H.H. Hosmer. Metapolicies II. In *Proceedings of the 15th National Computer Security Conference*, pages 369–378, 1992.
14. V. Issarny and C. Bidan. Aster: A Framework for Sound Customization of Distributed Runtime Systems. In *Proceedings of the Sixteenth IEEE International Conference on Distributed Computing Systems*, 1996.
15. V. Issarny, C. Bidan, and T. Saridakis. Designing an open-ended distributed file system in Aster. In *Proceedings of the 9th International Conference on Parallel and Distributed Computing Systems*, 1996.
16. V. Issarny, C. Bidan, and T. Saridakis. Customizing Middleware to Meet Quality of Service Constraints. Submitted for publication, 1997.
17. P. Janson and R. Molva. Security in open networks and distributed systems. *Computer Networks and ISDN Systems*, (22):323–346, 1991.
18. B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in Distributed Systems : Theory and Practice. *ACM Transactions on Computer Systems*, 10(4):265–310, November 1992.
19. C. E. Landwehr. Formal models for computer security. *ACM Computing Surveys*, 13(3):247–278, November 1981.
20. J. Magee, N. Dulay, and J. Kramer. A Constructive Development for Parallel and Distributed Programs. In *Proceedings of the International Workshop on Configurable Distributed Systems*, 1994.
21. J. McLean. The Algebra of Security. In *Proceedings of the 1988 IEEE Computer Society Symposium on Security and Privacy*, pages 2–7, April 1988.
22. National Computer Security Center. Trusted network interpretation of the tcsec. Technical Report NCSC-TG-005, July 1987.
23. OMG. The Common Object Request Broker: Architecture and Specification – Revision 2.0. Technical report, OMG Document, 1995.
24. OMG Security Working Group. White Paper on Security. TC Document 94.4.16, OMG, April 1994. Available by ftp at `ftp.omg.org:/pub/docs`.
25. D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.
26. J. M. Purtilo. The Polylith software bus. *ACM Transactions on Programming Languages and Systems*, 16(1):151–174, 1994.
27. B. Schneier. *Applied Cryptography, Second Edition: Protocols, Algorithms and Source Code in C*, volume ISBN 0-471-11709-9. John Wiley & Sons, Inc., 1993.
28. M. Shaw, R. DeLine, D. Klein, T. Ross, D. Young, and G. Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, 21(4):314–335, 1995.
29. Sun Microsystems Inc. The Java Virtual Machine Specification. Technical report, Sun Document, 1995.
30. L. van Doorn, M. Abadi, M. Burrows, and E. Wobber. Secure Network Objects. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 211–221, May 1996.
31. E. Wobber, M. Abadi, M. Burrows, and B. Lampson. Authentication in the Taos Operating System. In *Proceedings of ACM SIGOPS '93*, pages 256–269, 1993.
32. A. M. Zaremski and J. M. Wing. Specification matching of software components. In *Proceedings of the ACM SIGSOFT'95 Foundations of Software Engineering Symposium*, 1995.