A Formal Analysis of the CORBA Security Service

David Basin, Frank Rittinger, Luca Viganò

Institut für Informatik
Albert-Ludwigs-Universität Freiburg
Georges-Köhler-Allee 52
D-79110 Freiburg, Germany
Tel: +49 (0)761 203-8240, Fax: +49 (0)761 203-8242
{basin,rittinge,luca}@informatik.uni-freiburg.de
http://www.informatik.uni-freiburg.de/~{basin,rittinge,luca}

Abstract

We give a formal specification of the security service of CORBA, the Common Object Request Broker Architecture specified by the Object Management Group, OMG. In doing so, we tackle the problem of how one can apply lightweight formal methods to improve the precision and aid the analysis of a substantial, committee-designed, informal specification. Our approach is scenario-driven: we use representative scenarios to determine which parts of the informal specification should be formalized and then verify our formal specification against these scenarios. For the formalization, we have specified a significant part of the security service's data-model using the formal language Z. Through this process, we have been able to sharpen the OMG-specification, uncovering a number of errors and omissions.

Keywords: CORBA Security, Formal Methods, Specification, Z.

1 Introduction

In large, heterogeneous, distributed information systems, *middleware* is often used to provide a communication medium between different system components, which may run on different platforms. By acting as a mediator between the different components, and as a uniform interface for the developer, middleware provides access and location transparency and thereby creates the illusion of a single composite system. Various middleware systems exist, such as OMG's Common Object Request Broker Architecture (CORBA) [19], Microsoft's COM+ [23], and Sun's Enterprise Java Beans [28]. In this paper, we focus on CORBA and use the formal language Z to give a formal analysis of CORBA's security service.

The CORBA security service specification [20], given by the OMG, consists of more than 350 pages, detailing interfaces and using natural language and ad-hoc diagrams to describe the semantics of the interfaces. The overall specification is quite complex and this complexity stems from several factors. First, the problem itself is inherently complex. Any specification-conform implementation must achieve multilateral security guarantees for distributed object systems where the objects execute on different platforms in different environments. Second, the solution must be *loose* (or *open*). The OMG, a consortium whose members come from over 800 companies and organizations [18], does not define new technologies, but rather develops standards that encompass the technologies of its members. Hence, any specification of a standard by the OMG must be loose in the sense that it admits as models all implementations given by OMG members. This looseness is achieved, for example, by simply giving signatures for interfaces, where the use of general types does not constrain the values of possible implementations. Finally, the specification has the complexity inherent in any document designed by a political process.

Our research is motivated by the question of whether, despite this complexity and the informal nature of the specification of the CORBA security service, it is possible to reason about the security of systems built using this service. Moreover, whether one can do this in a *lightweight* way [10] without plunging into the black hole of formalizing both CORBA and its security service in their entireties.

What we have accomplished is to formalize part of the OMG-specification for this service, guided by particular security requirements. Our approach is scenario-driven: we use representative scenarios to back out which parts of the informal specification should be formalized, and then verify our formal specification against these scenarios by investigating whether the requirements are met in the scenarios. For the formalization, we have specified a significant part of the security service's data-model using the formal language Z. Through this process, we have been able to sharpen the OMG-specification and uncover a number of errors and omissions. Moreover, as Z supports high-level, readable specifications, we believe that our formal specification can also be of use to different vendors who wish to implement the CORBA security service or compare (or even formally verify) their implementation against the specification.

We have used tool support to improve the precision of our specification and the accuracy of our analysis. We wrote and type-checked our Z-specification using the ZETA-tool [33], which provided substantial assistance in finding errors in preliminary versions of our Z-specification and in the original OMG-specification. We initially carried out the analysis of the security properties by hand, and afterwards machine-checked many of our proofs, including the ones given here, using HOL-Z, which is an embedding of Z in higher-order logic within the generic theorem prover Isabelle [9, 14, 22]. In doing so, we have been able to show that some of the security requirements are satisfied, while for others a refinement

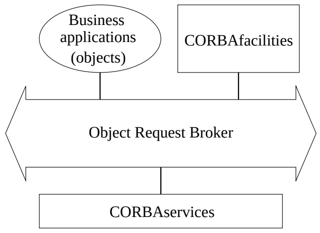

Figure 1: Object Management Architecture (OMA)

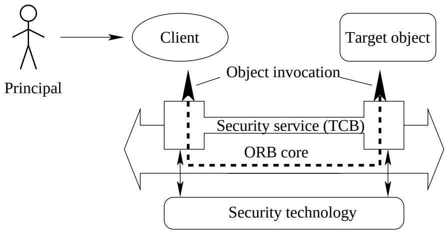

Figure 2: Integration of the security service

of our (and thus of the OMG's) specification is needed.

More generally, our work contributes to the understanding of how formal languages like Z can be used to model and analyze complex, informal standards. A common misconception is that formal methods are too heavy in such cases, as they require a complete formalization of the standard, which is of overwhelming complexity due to its size and the need to integrate different views of the system, e.g. static data-models with dynamic event or process-oriented models. We show here that it is possible to proceed with a pragmatic "light touch" in the formalization: One can proceed in a scenario-driven way, and even restrict attention to a particular view of the system (namely the data-view), and still arrive at general, useful results, with limited resources.

The rest of this paper is organized as follows. In § 2, we provide background on CORBA and its security service. In § 3, we discuss, and illustrate by means of examples, our Z-specification of this service. In particular, we consider parts relevant for authentication and access control. In § 4, we provide highlights from the formal analysis of our Z-specification. We conclude in § 5, where we summarize the lessons learned from our research and discuss related and future work.

2 Background on CORBA

2.1 OMG's CORBA

CORBA systems are based on the *Object Management Architecture* (OMA) [21, 24], which consists of an object model that describes the properties of all objects, and a reference model that describes how these objects interact. Figure 1 shows the four different types of components distinguished by the OMA: the *business applications* (*objects*), the *services*, the *facilities*, and the *Object Request Broker* (ORB). The ORB acts as an object bus (analogous to a hardware bus) that connects the different system components and manages communication between them, thereby ensuring portability, reusability and interoperability. The services provide general interfaces such as transaction management, object location and security, while the facilities provide domain specific interfaces (e.g. for health or financial services). This enables a kind of plug-and-play mechanism. Business applications only implement the business logic and are associated with services or facilities to gain additional functionality.

To achieve platform independence, all components in a CORBA system are known only through their interfaces, which are defined in a programming language independent way using the CORBA-specific *Interface Definition Language* (IDL). The OMG-specification, however, does not always have a clear semantics of what functionality implementations of the interfaces must provide; what semantics there is, is given only by informal prose or ad-hoc diagrams. As a result, the OMG-specification is ambiguous in many places, and this problem is complicated by the desire to have a specification open to all member technologies.

We focus here on version 1.2 of the CORBA security service [20]; for more information in general on CORBA consult [19, 21, 24], and [2, 5] for security in particular.

2.2 The CORBA Security Service

There are many different goals that must be achieved to obtain a secure system, and in this paper we focus on *confidentiality* and *integrity*. Together these two goals express that only authorized users can access and modify confidential data.

The protection mechanisms that are employed by the CORBA security service to achieve these goals are:

Identification and authentication: A user's identity is validated, e.g. by name and password, by biometry, or by similar mechanisms.

Authorization and access control: An authenticated user is assigned rights and is only allowed to perform actions for which he has rights.

It is mandatory for security that these protection mechanisms are always used and cannot be bypassed.

The CORBA security service specification defines interfaces between standard security technologies (such as Kerberos or SSL) and the ORB or the business applications. The architecture of the security service is based on a *Trusted Computing Base* (TCB), which encapsulates all security-related functionality in a small trusted core that cannot be bypassed. Given this encapsulation, the business components can then be implemented independently of concrete security mechanisms. This has the advantage that the code implementing the security functionality is minimized, security policies are easier to keep consistent, and the trust between different components is limited to the trust between the TCBs of different domains. Figure 2 illustrates how the security service is integrated into the ORB.

The actors in a CORBA environment, which are normally human users but can also be software or hardware components, are called *principals*. Principals are known to the system through *security attributes*, which are divided into *identity attributes* and *privilege attributes*. These attributes are assigned to a principal on registration, during which the principal is authenticated and is given its *credentials* (which include the principal's attributes).

Since CORBA is an object system, all actions in the system are carried out through object invocations, whereby the access control mechanisms must check the identity and rights of the initiating principals. The CORBA security model does not commit to one access model but allows designers to choose between many different models, such as access control lists, label-based policies or capabilities-based ones.[1]

3 Z-Specification of the CORBA Security Service

3.1 The Formal Language Z

Z [11, 30] is a formal language based on typed set theory and first-order logic with equality. It has been successfully applied, in its original form or using extensions such as Object-Z [29, 31], in many case studies in formal modeling and development, e.g. [6, 7, 15, 25].

We have chosen Z for our work as the CORBA security service specification is heavily data-oriented in that approximately half of it is focused on data, interfaces, and their functional requirements (while the remainder focuses on inter-ORB communication). The Z language is well-suited for such data modeling. In particular, Z provides constructs for structuring and compositionally building data-oriented specifications: schemas are used to model the states of the system (*state schemas*, e.g. the schema *Env* in § 3.4) and operations on the state (*operation schemas*, possibly with input/output parameters, as in the schema *authenticateSuc* in § 3.5), and a *schema calculus* is provided to compose these subspecifications.

We have also chosen Z for its simplicity: Z-specifications are fairly easy to write and understand. This is important for us as we would like the results of our formal analysis to be meaningful to those working with middleware who are not trained in formal methods. Another reason for choosing Z is its rapidly growing tool support. The Z-specification we describe here has been written and type-checked using the ZETA-tool [33] and we have begun to machine-check our proofs using the HOL-Z embedding in the generic theorem prover Isabelle [9, 14] (we return to this in § 4 below).

In what follows, we will assume that the reader is familiar with the syntax and features of Z. We will only explain a few details, as need arises, in the examples; a more complete picture of Z can be found in [11, 30].

3.2 Our Specification Approach

Given the size and complexity of the informal CORBA security service specification, we sought an alternative to analyzing the complete service. We approached the formalization problem by selecting scenarios for two of the most important security requirements: principal authentication and secure object invocation (including establishing security associations and access control).

From these scenarios we back out two kinds of specification. First, as presented in this section, we identify what parts of the OMG-specification must be formalized to describe those parts of the security service relevant for each scenario. Second, as presented in § 4, we back out the scenarios' security requirements, which constitute a (partial) specification of properties of the security service. These specifications form the basis for formally showing (in § 4) that the security service specification has the desired properties.

Due to space restrictions, we cannot give here in full our Z-specification of the CORBA security service, which encompasses more than 50 pages and can be found in [26]. Instead, we provide only a few highlights of our specification and illustrate the methodological aspects that lie at the heart of our approach. Note also that most of the Z-definitions (types, schemas, requirements) in this paper have been significantly simplified to include only those parts of the specification we actually focus on here.

Our translation of the informal OMG-specification of the CORBA security service into a Z-specification proceeded in three steps, where we actually translated into Z only those parts of the informal specification that were needed for the scenarios that guided us.

In the first step, we defined a mapping from IDL-constructs to Z-constructs.[2] Although it is possible to map each IDL-construct following a fixed scheme, it leads to a clearer and more concise formalization to translate IDL-constructs differently, according to their purposes. For instance, the canonical way to transform operations is to model them as operation schemas, but we can obtain a more concise specification if we model the operations that do not modify the state (i.e. lookup operations) as functions in Z. Similarly, we did not explicitly define a state schema for each of the interfaces, since this is only needed when the interface comprises state information. In the second step, we transformed all IDL-code into Z, and in the third step we augmented our Z-definitions with the informal descriptions of the interfaces given by the OMG.

The last two steps leverage off the fact that a Z-schema consists of two parts: a *declaration* part, where typed variables are declared, and a *predicate* part, where the constraints on the variables are specified. The second step of our translation amounts to mapping IDL-code into the declaration parts of schemas. The predicate parts, defined in step three, allow us then to rigorously specify the intended semantics, which is only informally described by the OMG.

In the rest of this section, we motivate and explain the scenarios we chose, provide some of the basic definitions that are needed to understand our specification, and define a formal model of the scenarios.

3.3 Scenarios

As examples, we consider here two requirements that we can impose on the specification, namely (i) that upon authentication, no principal can gain more attributes (i.e. more credentials) than the system allows, and (ii) that a valid security association must be established prior to a secure object invocation. To analyze these two requirements we consider the following two scenarios.

Scenario 1: Principal authentication. Whenever a principal enters a CORBA system for the first time, the principal must authenticate itself to the system, after which (if the authentication has been successful) the system assigns identity and privilege attributes to the principal.

Scenario 2: Security association. Before a client can invoke any operation on a target object, there must exist a security association between them. Prior to object invocation, the CORBA environment establishes this association, which is used to establish mutual trust between the client and the target and to negotiate the details of the communication. The process of establishing this association (also known as 'binding') is handled internally and is transparent to the client.

[1] Some of these models have been formally specified in [3, 13]; as a comparison, note that these works specify single access control models of CORBA while the OMG specification of the CORBA security service, and thus our specification in Z, defines how access control is exercised.

[2] Our mapping, which is defined for the IDL-constructs occurring in the CORBA security service specification of the OMG, has been influenced by the similar mapping given in [16], and could be automated (at least partially) using a compiler similar to the one given there.

3.4 Basic Definitions

Before we can formalize the interfaces associated with our scenarios, we must define the system environment. This is accomplished by the schema *Env* below, whose declaration part defines: the system time $systemTime$ ($UtcT$ is a time format defined by the OMG in the CORBA time service and is used in all CORBA services); a set $supportedAuthMethods$ of supported authentication methods (the authentication method chosen by the client must be supported by the actual security service implementation); a set *moreSteps* that consists of all authentication methods that require more than one step for authentication; a set *principals* of all registered users of the system; a function *attributes* that maps users to their security attributes; and a function *authenticates* that maps registered principals to their authentication data (*Opaque* is an unstructured sequence of bytes that is used throughout the OMG-specification of the security service when no assumptions about the data are made).

The predicate part of the schema *Env* specifies that the default attribute for each unregistered principal ($s \notin principal$) is equal to *Public*, thereby allowing public access to non-confidential data (if desired).

$$
\begin{array}{|l}
\text{— } Env \text{ —}\\
systemTime : UtcT\\
supportedAuthMethods : \mathbb{P}\, AuthenticationMethod\\
moreSteps : \mathbb{P}\, AuthenticationMethod\\
principals : \mathbb{P}\, SecurityName\\
attributes : SecurityName \rightarrow \mathbb{P}\, SecAttribute\\
authenticates : principals \rightarrow Opaque\\
\hline
\forall s : SecurityName \mid s \notin principals \bullet\\
\quad \exists a : SecAttribute \bullet\\
\quad\quad a.attribute\_type.attribute\_type = Public\\
\quad\quad \wedge\ attributes(s) = \{a\}
\end{array}
$$

As we mentioned above, most of the Z-definitions we give here are incomplete as we have, where possible, omitted several details for brevity and simplicity; the complete definitions can be found in [26]. Note also that the types in the OMG's, and our, specification can be divided into two kinds: (i) basic types that, according to the OMG, are given meaning only by their names, e.g. *SecurityName*; and (ii) complex types such as enumeration types (e.g. $AuthenticationMethod$) or structured types (e.g. *SecAttribute*). Whenever not relevant for our examples, we do not specify the details of these complex types. Moreover note that some of the identifiers may look needlessly complex, e.g. $a.attribute\_type.attribute\_type$, but they reflect the way the OMG structures its specification and the names chosen there.

In addition to the environment, we also require definitions for some of the basic entities occurring in the specification. As mentioned in § 2.2, all principals are identified by their attributes, which are managed in their credentials. These credentials are described by a schema *Credentials*, which defines components for the principal's attributes (*attributes*) that are split into privilege attributes (*privileges*) and identity attributes.

$$
\begin{array}{|l}
\text{— } Credentials \text{ —}\\
privileges, attributes : \mathbb{P}\, SecAttribute\\
\hline
privileges \subset attributes\\
\forall p : privileges \bullet\\
\quad p.attribute\_type.attribute\_family.family = 1
\end{array}
$$

The predicate part of the schema *Credentials* states that the privileges are a strict subset of all attributes, namely the attributes whose family is equal to 1 (attribute families can be used by third parties to extend the security service with new security attributes for special purposes that are not defined by the OMG service specification).

The following state schema *CredentialsSystem* describes an object that manages all credentials. It includes[3] the schema *Env* to access the system time, it defines a function *credentialsId* that maps credentials identifiers (of basic type *CREDS*) to credentials, and it defines a relation is_valid that pairs each valid credentials identifier with the time t when it expires, whereby t must be greater than the current system time.

$$
\begin{array}{|l}
\text{— } CredentialsSystem \text{ —}\\
Env\\
credentialsId : CREDS \nrightarrow Credentials\\
is\_valid : CREDS \leftrightarrow UtcT\\
\hline
\forall c : \mathsf{dom}\, credentialsId;\ t : UtcT \mid (c, t) \in is\_valid \bullet\\
\quad t > systemTime
\end{array}
$$

The state schema *Current* describes context and technology-specific state information for the current principal, such as the credentials that will be used for the next invocation.

$$
\begin{array}{|l}
\text{— } Current \text{ —}\\
own\_credentials : \mathrm{seq}\, CREDS
\end{array}
$$

3.5 Scenario 1: Principal Authentication

The CORBA security service specification defines the authentication of principals in the interface *PrincipalAuthenticator*. This interface defines two operations: *authenticate* to authenticate a principal in one step, and $continue\_authentication$, which specifies the behavior of the system when additional authentication steps are required. (When such steps are necessary, and how they should be processed, is technology-dependent and cannot be analyzed without knowledge of the underlying technology. In particular, it is impossible to define a schema describing the operation $continue\_authentication$ without this knowledge.)

We focus on the case where only one step is needed for the authentication and use the schema $authenticateContinue$ to represent the further steps for the authentication and the conditions under which these steps occur. Indeed, we exploit the schema calculus of Z to define the operation *authenticate* as a disjunction of several schemas capturing successful, partial or (in the last three disjuncts) erroneous authentication:

$$
\begin{array}{l}
authenticate == authenticateSuc\\
\quad \vee\ authenticateContinue \vee methodsNotSupported\\
\quad \vee\ wrongAuthData \vee tooManyAttributes
\end{array}
$$

As an example, the operation schema *authenticateSuc*, for successful authentication, is as follows:

[3] Inclusion of a schema S_1 in a schema S_2 means that the declaration part of S_1 is incorporated into the declaration part of S_2 and the constraints on these declarations imposed in the predicate part of S_1 are propagated into the predicate part of S_2. Other forms of schema inclusion are used and explained below.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{authenticateSuc}} \\
\hline
\Xi Env \\
\Delta Current \\
\Delta CredentialsSystem \\
method? : AuthenticationMethod \\
security\_name? : SecurityName \\
auth\_data? : Opaque \\
privileges? : \mathrm{seq}\, SecAttribute \\
creds! : Credentials \\
return! : AuthenticationStatus \\
\hline
method? \in supportedAuthMethods \\
method? \notin moreSteps \\
auth\_data? = authenticates(security\_name?) \\
\mathsf{ran}\, privileges? \subseteq attributes(security\_name?) \\
\exists\, c : Credentials;\ cid : CREDS;\ t : UtcT \bullet \\
\quad cid \notin \mathsf{dom}\, credentialsId \land c \notin \mathsf{ran}\, credentialsId \\
\quad \land\ credentialsId' = credentialsId \cup \{cid \mapsto c\} \\
\quad \land\ is\_valid' = is\_valid \cup \{cid \mapsto t\} \\
\quad \land\ creds! = c \land c.privileges = \mathsf{ran}\, privileges? \\
\quad \land\ own\_credentials = \langle cid \rangle \\
return! = SecAuthSuccess \\
\hline
\end{array}
$$

This schema describes the possible transitions of the part of the system state that is represented by the schemas *Current*, *CredentialsSystem* and *Env*. To this end, *Current* (and similarly for *CredentialsSystem*) is included prefixed by a Δ, meaning that two copies, primed and unprimed, of all variables of *Current* are included in *authenticateSuc*; unprimed variables represent the state before the transition, while primed variables define the state after it. Inclusion of a state schema prefixed by Ξ, like for *Env* in this case, works similarly but with the restriction that the values of the variables remain unchanged during the transition, i.e. the values of primed and unprimed variables must be equal. Overall, the predicate part of *authenticateSuc* describes the conditions for successful authentication, and thereby formalizes the semantics of this operation.

A successful authentication requires that the chosen authentication method *method?* is supported by the environment and that, for a one-step authentication, *method?* is not an element of the set *moreSteps*. Furthermore, the correct authentication data for the given security name *security_name?* is checked against the provided data *auth_data?*, and the privileges requested by the client (*privileges?*) must be contained in the attributes the client is entitled to. The appropriate credentials (including the requested privileges) are then created, added to the set of valid credentials, and set in the client's *Current*.

To capture the scenario when the authentication is refused by the system because the provided authentication data does not match the required data, we define the schema *wrongAuthData*. Its precondition states that the authentication data is incorrect and therefore indicates failure of the authentication ($return! = SecAuthFailure$) in contrast to *authenticateSuc*. No state changes are described by this schema.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{wrongAuthData}} \\
\hline
\Xi Env \\
security\_name? : SecurityName \\
auth\_data? : Opaque \\
return! : AuthenticationStatus \\
\hline
auth\_data? \neq authenticates(security\_name?) \\
return! = SecAuthFailure \\
\hline
\end{array}
$$

The other schemas that indicate failures, *methodsNotSupported* and *tooManyAttributes*, are defined analogously.

3.6 Scenario 2: Security Association

A security association between communicating parties is employed to establish trust between them and to negotiate the details of the communication, such as the type and strength of protection used for the messages they exchange. The state/context information of this association is represented on both sides by a *SecurityContext* interface, through which the target object has access to the credentials of the initiator of the invocation.

The Z-formalization of this interface is analogous to that of the *Credentials* interface. To represent the state information of the context, we define a schema *SecurityContext*, which defines variables for the credentials, for the target object, and for additional context-specific information (which we do not display in the schema). The state schema *ObjectSystem* is included since it defines components that manage the objects that are currently associated with the system; in particular, it defines a function *objectId* that maps object identifiers to the actual objects.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{SecurityContext}} \\
\hline
ObjectSystem \\
credentials : \mathbb{P}\, CREDS \\
target : \mathsf{dom}\, objectId \\
\hline
\end{array}
$$

A security context can only be created through the operations *init_security_context* and *accept_security_context*, which are defined in the interface *Vault*. But since we are only concerned here with the existence of such contexts, we elide in our examples the parts of the specification that handle their establishment.

The schema *SecurityContextSystem* describes an object that manages all *SecurityContext* objects (i.e. it describes the state of the system of all client and target contexts). It defines a function *securityContextId* that maps identifiers to objects and a relation *is_valid* that pairs the identifiers of all valid objects with their expiration date. In addition it is necessary to define two sets of contexts, one for the client-side contexts and one for the target-side contexts. In the predicate part, these two sets are constrained to be of the same size and to contain all valid contexts in the system.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{SecurityContextSystem}} \\
\hline
\Xi Env \\
securityContextId : SEC\_CONTEXT \rightarrowtail\!\!\!\!\!\!\!\!\!\;\;\;\; SecurityContext \\
is\_valid : SEC\_CONTEXT \leftrightarrow UtcT \\
clientContexts : \mathbb{P}\, SEC\_CONTEXT \\
targetContexts : \mathbb{P}\, SEC\_CONTEXT \\
\hline
clientContexts \cup targetContexts = \\
\quad \mathsf{dom}\, securityContextId \\
\#clientContexts = \#targetContexts \\
\forall\, s : SEC\_CONTEXT;\ t : UtcT \mid \\
\quad (s, t) \in is\_valid \bullet t > systemTime \\
\hline
\end{array}
$$

The next section will show that, although abstracting away details about how the security association is established, these two schemas are sufficient to describe the part of the security service against which we will check the requirements of this scenario, i.e. the existence of two corresponding *SecurityContext* elements.

4 Formal Analysis: Two Examples

We have carried out a formal analysis of our Z-specification of the CORBA security service by using our scenarios to identify security requirements, and afterwards formalizing them in Z and attempting to prove (using HOL-Z [9, 14]) that they hold of our specification.

Since we are only using Z for data-modeling, some requirements fall outside of our formal specification and analysis. In general, such an analysis yields three different kinds of results:

(i) the specification adheres to the requirement, which follows by proving a formula that represents the requirement;

(ii) the specification does not adhere to the requirement, but may do so after the specification is refined; and

(iii) the requirement cannot be expressed using our formalization in the first place.

The two examples that we present below illustrate these three situations; further examples are given in [26].

A Formal Analysis of Scenario 1

The first example of our formal analysis, which illustrates results of kinds (i) and (iii), is to investigate whether principal attributes are assigned correctly upon authentication. Namely, we require from our Z-specification that principals are always authenticated and are assigned correct credentials.[4]

We can split this requirement into two subrequirements, the first of which can be formulated as: The operation *authenticate* must be called for the initiating principal prior to any object invocation on behalf of that principal. This subrequirement cannot be directly expressed in our Z-specification by formalizing properties about schemas in the specification that specify state or operations that transform state. Capturing these kinds of temporal dependencies is best accomplished by combining Z with another specification language, like a temporal logic, or some visual, temporally-oriented specification formalism like UML's sequence diagrams, where one can express constraints on the *dynamics* of the system, e.g. which sequences of operations are allowed. The importance of such combinations is discussed in more detail in § 5.

Many of the security requirements can, however, be expressed as conditions on the data-model. Consider, for example, the second subrequirement, which states that the credentials assigned to a principal during authentication are correct, in the sense that no privilege attributes are assigned to a particular principal other than those that the system allows for him. We can formalize this in Z as follows:

$$\forall\, Env;\ Current;\ CredentialsSystem \bullet authenticate \Rightarrow creds!.privileges \subseteq attributes(security\_name?) \qquad (1)$$

Here, quantification over the schemas *Env*, *Current*, and *CredentialsSystem* quantifies over the free variables in the declaration parts of these schemas, which binds the variables in the body of (1). The schemas *Env*, *Current*, and *CredentialsSystem* denote the system state for our example and the quantification formalizes that property (1) must hold for all possible states.

We are now in a position to prove that the specification adheres to this subrequirement, i.e. that (1) holds. To do so, we must show that, for an arbitrary system state (represented by *Env*, *Current*, and *CredentialsSystem*), the operation *authenticate* only returns credentials that are in accordance to the system's policy for the initiating principal. Note that it is only necessary to show this for a successful authentication, as formalized by *authenticateSuc*, since otherwise no credentials are created and set in the principal's *Current*. The essence of the proof is then as follows: one of the conditions in the predicate part of *authenticateSuc* states that

$$\mathsf{ran}\, privileges? \subseteq attributes(security\_name?)$$

and $c.privileges = \mathsf{ran}\, privileges?$ (also in the predicate part of *authenticateSuc*) implies that

$$c.privileges \subseteq attributes(security\_name?)\,,$$

so that $creds! = c$ yields the requirement

$$creds!.privileges \subseteq attributes(security\_name?)\,.$$

As the instances of *ORB* and *authenticate* were arbitrary, we can conclude with (1).

A Formal Analysis of Scenario 2

The second requirement that we consider concerns the security association for a secure object invocation, and is an example of a result of kind (ii) illustrating how formal reasoning can lead to an improvement of the formal specification. In order to formulate the requirement, we briefly explain the access control mechanisms in the CORBA security service.

Since CORBA is an object system, to control access to confidential data one must control the invocation of operations on objects. Access control is realized by an operation *access_allowed* that checks the rights of a principal (which is represented through its credentials) to invoke an operation on an interface. The operation *access_allowed* is specified by a relation that contains all tuples of credentials, operation names, and interface names for which access is allowed, i.e. if the tuple $(c, t, op, i) \in access\_allowed$ then the principal with credentials c is allowed to invoke the operation op on the target object t implementing the interface i.

As far as the security service is concerned, prior to an access decision for an operation invocation of a client object on a target object, a security context must have been established between the client and the target.

We formulate the second security requirement as: A client can only invoke an operation on a target if a security association between them has been previously established. More technically, the following formula states that a valid client context must be present in the *clientContexts* component of the state schema *SecurityContextSystem* for each tuple in *access_allowed* before an invocation can occur. Since the actual object invocation is not part of the security service, we can assume that the last action of the security service prior to the invocation is a call of *access_allowed* for an access decision, and that the ORB handles this correctly. Therefore *access_allowed* will be used as a synonym for the actual invocation and, since *access_allowed* is modeled as a relation, the test for membership corresponds to the operation call. Moreover, we require that all security contexts and credentials are valid with respect to the current system time.

[4] It is important to note that in this example we are only concerned with the initial assignment of credentials, and therefore privileges, to principals. Since the security service provides operations for a client/principal to copy and modify its credentials, it is another proof obligation (not discussed in this paper) to show that these operations cannot be used to modify a principal's privileges illegally.

$$\exists\, scs : SecurityContextSystem;\ cs : CredentialsSystem;$$
$$sc : scs.clientContexts;\ cl : \mathrm{seq}\, CREDS;\ t : UtcT;$$
$$i, op : Identifier \bullet$$
$$(\mathsf{ran}\, cl \subseteq (scs.securityContextId\ sc).credentials$$
$$\land\ (sc, t) \in scs.is\_valid$$
$$\land\ (\forall\, c : cl \bullet \exists\, t' : UtcT \bullet (c, t') \in cs.is\_valid))$$
$$\Leftrightarrow (cl, (scs.securityContextId\ sc).target, op, i)$$
$$\in access\_allowed$$

Additionally to the formula above, we must also require that an element exists in *targetContexts* and that it is connected to the one in *clientContexts*, i.e. that they were created by consecutive calls to *init_security_context* and *accept_security_context*.

Our use of formal methods helped us uncover a problem here. Namely, we discovered that we could not formulate this additional requirement since the connection between the elements in *targetContexts* and *clientContexts* is not present in the specification of the OMG and, a fortiori, in ours. Although the OMG-specification in its current form is inadequate, it is possible to extend it, and, correspondingly, refine our Z-specification, so that we can then state and prove the requirement. This is achieved by augmenting the schema *SecurityContextSystem* with a relation *associates* that relates a client-side security context object with the corresponding target-side object.

$$|\ associates : SEC\_CONTEXT \leftrightarrow SEC\_CONTEXT$$

Having repaired this omission, we can now check that a pair of valid contexts is contained in this relation when access decisions are made. That is, we can now establish that when a security service meets the specification given by the *SecurityContext* schema and the extended *SecurityContextSystem* schema then it satisfies the requirements of our second scenario.

5 Conclusions

We now summarize the lessons learned and discuss related and future work.

Lightweight formal methods. As explained in the introduction, the informal specification given by the OMG is large and complex. A number of groups have carried out partial, informal analyses of the CORBA security service [1, 17], but the task of a full formal analysis is rather daunting. Our work indicates that there are reasonable points in-between when formal methods are used in a pragmatic, lightweight fashion.

To carry out a formal analysis, we have reduced the complexity of the problem along two dimensions. First, we took a data-oriented view of the system and formalized the service's data-model in the Z language, and second we took a scenario-driven approach to determine which parts of the OMG-specification should be formalized, and then verified our formal specification against requirements of these scenarios. This reduction allowed us to give a partial, but meaningful, formal analysis, without formalizing both CORBA and its security service in their entireties, or managing the additional complexity of employing multilanguage formalizations.

Our formal specification. OMG-specifications are informal and open by design. This is politically attractive (since it subsumes the reference implementations of OMG members), but problematic. Although loose specifications (allowing model/implementation classes) are successfully used, e.g. in algebraic specification, when carried out using natural language, by a political process, too much "looseness" creeps in. This allows future developers and implementors of the security service to interpret the specification in unintended ways that may lead to security-holes in applications built using the service. A formalization of the specification (especially in a language that is relatively easy for non-specialists to understand) helps solve this problem, as it provides a rigorous semantics that can serve as a basis for communication between different developers, implementors and users of the system. Even a partial formalization is of assistance here, e.g. clarifying the data-model, although leaving control semi-formal. Moreover, even with a partial formalization, it is still possible, and becomes interesting, to formally establish security properties and thus demonstrate that the specification is "tight" enough.

In the formal specification process (and even more so in the formal analysis process), our experience indicates that appropriate tool support is a necessity. Tool support improves the understanding of, and increases the confidence in, the specification and analysis. Indeed, it is difficult to write a Z-specification of more than a handful of schemas without any typographical errors or inconsistencies. The use of the type-checker ZETA [33] enabled us not only to correct many such mistakes in preliminary versions of our Z-specification but also to discover a number of errors and omissions in the original OMG-specification.

Our formal analysis. A formal specification like ours provides the basis for a formal verification of many of the multilateral security properties of the security service and of applications built upon it. Rather than completely analyzing the entire service, a Herculean task due to its complexity, we adopted a scenario-driven analysis and, as examples, focused in this paper on two security scenarios: principal authentication and secure object invocation. (Other scenarios and requirements are analyzed in [26].) Our examples illustrate how our analysis yields three different kinds of results: (i) the Z-specification adheres to the requirement; (ii) the Z-specification does not adhere to the requirement; and (iii) the requirement cannot be expressed using our formalization in the first place.

Our proofs of results of the first two kinds were carried out by hand and also machine-checked using the HOL-Z embedding in the generic theorem prover Isabelle [9, 14]. Results of kind (iii), on the other hand, identify a shortcoming of our approach.

Z and other formal methods. Our examples show how the use of Z can improve the precision and analysis of an informally specified standard like the CORBA security service. As mentioned above, Z is well-suited for data-oriented modeling, e.g. for modeling the states of the system and the operations on them, and hence well-suited for our approach where we define such a model. The decision to focus on the data-model bounds the overall complexity, but has, as its main drawback, the limitation that there are security scenarios and properties that fall outside of this model as they require formalizing other views, for which other formal methods are much better suited than Z. Examples include event-oriented or process-oriented views like those formalizable using process algebras like CSP [27] or temporal logics, as was done in [4, 12], where the model-checker SPIN is used to validate parts of the CORBA specification. These methods allow one to express explicit temporal constraints and analyze sequences of actions for safety and liveness properties, which is important for security, since it may happen that different components or operations taken by themselves are well-defined but their correct use requires their proper sequencing with events from the environment. As future work, we plan to analyze not only further scenarios within the data-oriented view, but also other views by enhancing and combining our approach with event-oriented or process-oriented formalisms. Existing embeddings of such formalisms, such as the HOL-CSP embedding in Isabelle [8, 32], will provide a good starting point, with HOL serving as a common semantic basis for combining our Z-specification with a process-oriented CSP-specification.

Important related work is also that involving the use of Object-Z [29], which combines object oriented and temporal extensions to Z. We strongly considered the use of this language for our work, especially since it has been applied in case studies of specifications of CORBA services [15, 25]. In the end, we chose Z for two reasons. First, Z is an ISO standard, which carries weight with standardization bodies like the OMG. Second, and more importantly, Z is supported by many tools for modeling and reasoning, such as ZETA and HOL-Z, which provided a basis for our work.

References

[1] J. Arndt and T. Österdahl. Network security in distributed systems using CORBA. Technical report, Ericsson Hewlett-Packard Telecommunications AB, 1998.

[2] B. Blakley. *CORBA Security.* Addison-Wesley-Longman, 1999.

[3] G. Brose. A typed access control model for corba. In F. Cuppens, Y. Deswarte, D. Gollmann, and M. Waidner, editors, *Proceedings of ESORICS 2000*, LNCS 1895, pages 88–105. Springer-Verlag, Berlin, 2000.

[4] G. Duval. Specification and verification of an object request broker. In *Proceedings of the 20th International Conference on Software Engineering (ICSE'98)*, pages 43–52. IEEE Computer Society Press, 1998.

[5] D. Gollmann. *Computer security.* John Wiley & Sons, 1999.

[6] J. Hall. Using Z as a specification calculus for object oriented systems. In D. Bjorner, C. A. R. Hoare, and H. Langmaack, editors, *Proceedings of VDM'90*, LNCS 428, pages 290–318. Springer-Verlag, 1990.

[7] I. Hayes. *Specification Case Studies.* Prentice-Hall International, 1993.

[8] HOL-CSP. `http://www.informatik.uni-freiburg.de/~wolff/doc/isa_doc/CSP/`.

[9] HOL-Z. `http://www.informatik.uni-freiburg.de/~wolff/doc/isa_doc/HOL-Z/home.html`.

[10] D. Jackson and J. Wing. Lightweight formal methods. *IEEE Computer*, 1996.

[11] J. Jacky. *The way of Z.* Cambridge University Press, 1997.

[12] M. Kamel and S. Leue. Validation of the General Inter-ORB Protocol (GIOP) Using the SPIN Model-Checker. Technical Report CWC03, Department of Electrical and Computer Engineering University of Waterloo, 1998.

[13] G. Karjoth. Authorization in CORBA Security. *Journal of Computer Security*, 8(2–3):89–108, 2000.

[14] Kolyang, T. Santen, and B. Wolff. A structure preserving encoding of Z in Isabelle/HOL. In J. von Wright, J. Grundy, and J. Harrison, editors, *Proceedings of TPHOLs'96*, LNCS 1125, pages 283–298. Springer-Verlag, Berlin, 1996.

[15] D. Kreuz. Formal specification of CORBA services using Object-Z. In *Proceedings of 2nd IEEE International Conference on Formal Engineering Methods (ICFEM'98).* IEEE Computer Society Press, 1998.

[16] D. Kreuz. *Formale Semantik von Konnektoren.* PhD thesis, Technische Universität Hamburg-Harburg, Germany, 1999.

[17] U. Lang. CORBA Security. Master's thesis, Royal Holloway University of London, 1997.

[18] OMG. `http://www.omg.org`.

[19] OMG. *CORBA/IIOP 2.2.* OMG, 1998.

[20] OMG Security Working Group. CORBAservices Security Service Specification v1.2. Technical report, Document: formal/98-12-17, OMG, 1998.

[21] R. Orfali, D. Harkey, and J. Edwards. *Instant CORBA.* Addison-Wesley-Longman, 1998.

[22] L. C. Paulson. *Isabelle: a generic theorem prover.* LNCS 828. Springer-Verlag, Berlin, 1994.

[23] D. Platt. *Understanding COM+.* Microsoft Press, 1999.

[24] A. Pope. *The CORBA Reference Guide.* Addison-Wesley-Longman, 1998.

[25] J.-C. Real. Object-Z specification of the CORBA repository service. Technical Report 351, Université Libre de Bruxelles, 1997.

[26] F. Rittinger. Formale Analyse des CORBA Sicherheitsdienstes. Diplomarbeit, Institut für Informatik, Albert-Ludwigs-Universität Freiburg, Germany, 2000. In German.

[27] A. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall International, 1997.

[28] G. Seshadri. *Enterprise Java Computing.* Cambridge University Press, 1999.

[29] G. Smith. *The Object-Z Specification Language.* Kluwer Academic Publishers, 1999.

[30] J. Spivey. *The Z Notation.* Prentice Hall International, 1998.

[31] S. Stepney. *Object Orientation in Z.* Springer-Verlag, 1990.

[32] H. Tej and B. Wolff. A corrected failure-divergence model for CSP in Isabelle/HOL. In J. Fitzgerald, C. Jones, and P. Lucas, editors, *Proceedings of the 4th Formal Methods Europe Symposium (FME'97)*, LNCS 1313, pages 318–337. Springer-Verlag, 1997.

[33] ZETA. `http://uebb.cs.tu-berlin.de/zeta/`.