University of Oslo
Department of Informatics

Towards a Framework of Authentication and Authorization Patterns for Ensuring Availability in Service Composition

Judith E. Y. Rossebø
Rolv Bræk

Research Report 332
ISBN 82-7368-287-0
ISSN 0806-3036

15th March 2007

Abstract

During the past decade, the telecommunication environment has evolved from single operator featuring voice services to multi-operator featuring a range of different types of services. Services are being provided today in a distributed manner in a connectionless environment requiring cooperation of several components and actors. This report focuses on the incremental means to ensure access to services for authorized users only by composing authentication and authorization patterns and services. We propose a novel framework of authentication and authorization patterns for securing access to services for authorized users only, and we demonstrate how the patterns can be dynamically composed with services using a policy-driven approach.

Contents

1 Introduction

The evolution of service development in the telecommunications sector, driven by the success of the Internet, creates a demand for dynamic service development in order to continuously develop new services in a competitive market. There is a need for fast incremental development of services and applications, while maintaining availability properties.

We define a service as an identified partial functionality, provided by a system, component, or facility, to achieve desired end results (goals) for end users or other entities. The general notion of a service involves several service parts collaborating to provide the service to one or more service users. Authentication and Authorization functionality needed to ensure availability is no exception and falls within this general definition of a service.

One of the core challenges of service engineering is to find practical ways to model services (partial functionalities) separately such that services may be composed into well functioning application systems satisfying availability requirements. This is especially challenging for services being provided in a distributed manner in a connectionless environment requiring cooperation of several components and actors (users).

If services were independent of each other, service composition would be quite straightforward. But services often depend on each other. Services also often depend on shared resources and service enablers. They may be provided to many interacting users, and users have access to many services over the same terminals using shared resources and service enablers. This leads to the so-called crosscutting nature of services as illustrated in Fig. 1.

The figure suggests an architecture for service-oriented systems, which is characterized by horizontal and vertical composition. On the horizontal axis, system components, are identified that are largely service independent and represent domain entities such as users, terminals, service enablers and shared facilities. They may reside in different computing environments. These domain entities such as users, user communities, terminals and resources are represented by agents in the system. We use the term agent in a general sense here to mean an entity representing and acting on behalf of other entities. On the vertical axis, several services and service components are identified (i.e. collaborations and collaboration roles) that depend on the system components of the architecture.

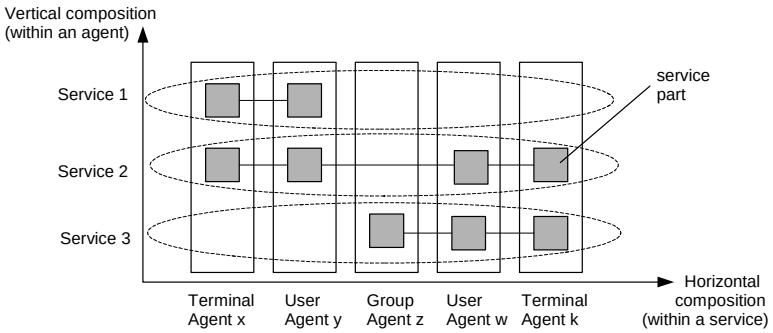

Figure 1: Service Oriented Architecture

Service composition, in general, involves static composition at design time as well

as dynamic linking and binding at runtime.

The new UML 2.0 collaboration concept [32] provides a structured way to define partial functionalities in terms of collaborating roles, and therefore it provides a promising basis for service modeling. It allows service parts to be modeled as collaboration roles, and service behavior to be specified using interactions, activity diagrams and state machines as explained in [43]. Moreover, it provides means to decompose/compose services using collaboration uses and to bind roles to classifiers defining system components. In this way, UML 2.0 collaborations directly support service modeling and service composition at design time. In addition, they provide a framework to define so-called semantic interfaces as explained in [46] that can be utilized to ensure compatibility among interacting components both at design time and runtime.

As Fig. 1 shows, service components interact with each other ("horizontally") for the actual execution of services. Services depend on each other (characterizing vertical composition) e.g., authentication and authorization behavior first, before a service can be invoked and services depend on shared resources and enablers (characterizing horizontal composition). The structure and linking of service components is to a large extent dynamic. Therefore, dynamic linking is a fundamental and general mechanism required in service-oriented systems. Important mechanisms for service discovery, feature selection, compatibility validation, and access control can be associated with the creation and release of dynamic links. This linking may be seen as a process of dynamically binding roles to actors, taking the agent states and preferences into account.

In [37], we have presented a conceptual model for service availability. Based on the conceptual model for service availability presented in [37], this report focuses on the incremental means to ensure access to services for authorized users only by composing authentication and authorization patterns and services. In order to address service availability, we see availability as a composite notion consisting of exclusivity, the ability to ensure access for authorized users only, and accessibility, the property of being on hand and useable when needed. Our approach involves the development of flexible and re-usable patterns to ensure availability in service composition.

In this report we motivate and introduce a set of authentication and authorization (AA-) patterns, which may be composed with services to ensure that services are accessible to the authorized users only. We provide a discussion of the specification of AA-patterns and the means to compose AA-patterns with services both statically and dynamically to restrict access to services to authorized users only. We explain in detail the policy driven approach to specifying composition of AA-patterns and services[1] .

In summary, our contributions include: (1) a framework and classification of authentication and authorization patterns; (2) demonstrating that our framework can be applied to static and dynamic composition; and (3) showing that our framework can be used to specify and enforce policies governing composition of AA-patterns and services.

The rest of this report is organized as follows: In Sect. 2 we state the requirements to the approach. In Sect. 3 we present our classification of authentication and authorization patterns. In Sect. 4 we discuss our approach to specifying AA-patterns, and in Sect. 5 we discuss how we apply policies. Use of AA-patterns in static composition of services is addressed in Sect. 6, and use of AA-patterns in dynamic composition of services is addressed in Sect. 7. A discussion of related work is given in Sect. 8 followed by a summary and conclusion in Sect. 9.

[1] We have published two articles based on this report [40,41]. However, since publication of these articles, some of the definitions regarding semantic interfaces have been refined [45]. This report has been updated to be consistent with the terminolgy presented in [45].

2 Requirements to the approach

In order to explain the policy-driven approach, we formulate and motivate a set of requirements that the approach is designed to fulfill.

1. *The approach should facilitate specification of authentication and authorization patterns in a flexible and reusable manner.* It should be possible to be able to model services independently and AA-patterns separately and then put them together and adapt them. As there are many different types of services, each with different authentication and authorization requirements, there is a need for a fine-grained approach that allows authentication and authorization design to be adapted to service requirements. The approach must allow for tailoring modifications with respect to e.g., strength of authentication provided and should be able to address the pitfalls that designers face in selecting and implementing authentication patterns to avoid faulty and weak implementations.

2. *The approach should be easy for a designer to understand and use.* Security requirements, such as availability requirements, are often not taken into account by developers in the design process for many reasons such as time to market and costs constraints, and lack of knowledge about security amongst designers and developers, as well as the complexity of the environment in which systems are deployed [3]. The approach should be understandable to the developer/designer and increase the designer's awareness of security issues while enabling the designer to address the issues systematically through choice and specialization of e.g. an authentication pattern.

3. *The approach should provide policy mechanisms that can be used for governing the binding of roles to agents in dynamic service composition.* This involves providing a means to specify constraints on the binding of roles to agents to ensure that service availability requirements can be achieved in a deployment. In order to ensure that service roles that are dynamically linked within a service execution are correctly linked, and to restrict which agents service roles can be bound to, there is a need for a means to defining rules to govern the binding of roles to agents.

4. *The approach should provide a means for specifying the static composition of AA-patterns and services.* Static composition involves the assignment and composition of roles to form system components at design time. This involves building composite services using existing services and AA-patterns (choreography), but also defining roles and system parts so that they can collaborate with each other (orchestration). The approach should provide a descriptive means to specify rules regarding ordering of collaborations in composition. In particular, requirements for which goals or states must have been achieved by collaborative parts before any other behavior is allowed to execute. The framework should also provide a means for modelling the dynamical restriction of the behavior involved in service composition, e.g. exceptions handling (such as in the case that authorizations are no longer valid, then the session should be forced to terminate).

5. *The approach should provide a means for supporting the dynamic composition of AA-patterns and services.* By dynamic composition, we mean both dynamic

role-binding (i.e., creation and release of dynamic links) and dynamic composition of service role behaviors.

3 Framework for AA-patterns

Our framework consists of a classification of authentication and behavioration techniques as patterns specified using UML 2.0 collaborations, with interactions and state machines. We also specify the AA-patterns using semantic interfaces [45], to facilitate validation of visible interface behavior for each of the roles involved in a collaboration and to enable dynamic composition of AA-patterns and services.

We specify behavior using semantic interfaces because semantic interfaces facilitate checking the compatibility (in terms of safety and liveness properties) of different components involved in service collaboration (Interface behaviors are derived from the complete component behaviors by projection). We declare role-binding policies in the semantic interface for each of the roles involved, as we find this useful for validating that the required conditions and requirements have been fulfilled when composing the pattern with services. In the following sections we present our classification of AA-patterns.

3.1 Authentication patterns

Authentication theory and practice has evolved over time and is well established in the literature [4,28,30] as well as in the standards [18–20,22,23]. The simplest authentication patterns involve two parties. Variations involve proxies, or trusted third parties. By a third party we mean a component, service or organization, which both other parties are willing to rely on. In some cases, each party relies on a different trusted third party, who in turn trust each other through a trusted third party. We begin therefore, by addressing patterns involving two parties, as these can be generalized or extended to involve trusted third parties.

A generic two party authentication pattern involves communication between the two parties to establish the identity of one of the parties in the case of unilateral authentication, or both in the case of mutual authentication. Messages are generated and exchanged between the parties, at least one message/pass is required for unilateral authentication, and at least two messages/passes are required for mutual authentication.

In order to apply authentication protocols and techniques [4, 28, 30] in a model-based approach, we have classified these well known authentication techniques and protocols as authentication patterns specified using UML 2.0 collaborations, which may be combined with service components in service composition. Each pattern is modelled in UML 2.0 so that it may be re-used, but also may be easily adapted and adjusted depending on requirements such as security and performance requirements, for example regarding the the strength of the crypto involved related to the capacity available in the actual deployment.

For modelling authentication of one actor playing one role in a service collaboration to an actor playing another role in a service collaboration we need a fine grained classification of authentication patterns. this is because the behavior required and strength of authentication required depends on the service to be deployed. In one case, unilateral one-pass authentication might be sufficient, e.g. for access to an online telephone catalogue, however, in another case, mutual two pass authentication may be required, as

is the case for authenticating terminals and access points to each other in third generation mobile networks. Additionally, in the first example, a simple message containing a username and password satisfies the service requirements, whereas in the second example, hardware protected keys for use in a symmetric-based crypto protocol are required. The behavior required, the type of keying to be used, and the strength of crypto to be used is service dependent.

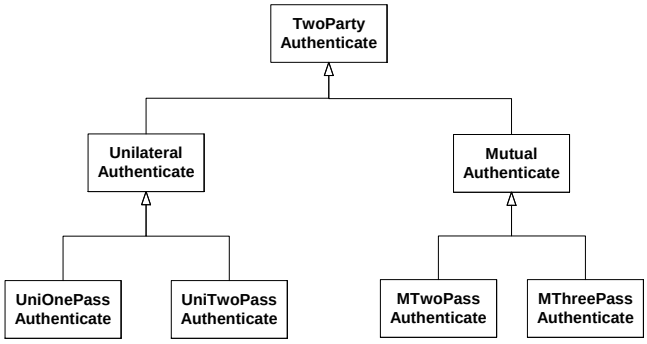

Figure 2: Authentication patterns

Our classification is therefore motivated by the need to address behavioral considerations in the patterns. This means classification based first on the service provided, unilateral authentication or mutual authentication, then based on the number of messages involved in the pattern, e.g., one message for a one pass authentication protocol, two messages for a two pass authentication protocol. Fig. 2 shows this generic classification of two party authentication patterns. For the full classification see Appendix C.

The aim is to make the developer more conscious in the choice of authentication technique to apply, while allowing flexibility with respect to the choice of protocol and algorithm(s) and other crypto techniques to be used. This allows the developer to focus on e.g. whether there is an issue such as timing regarding the number of messages involved e.g. one-pass, two-pass or three pass, or should symmetric or asymmetric keying be used, before choosing the protocol and algorithm in the instantiation of the pattern.

Fig. 2 shows these generic patterns that do not bind a particular protocol or algorithm. Once a generic pattern is selected, the authentication pattern can be further differentiated in specializing the pattern depending on the type of keying, e.g., symmetric or asymmetric, to be used.

For example, the unilateral one pass authentication pattern may be specialized as illustrated in the UML 2.0 class diagram shown in Fig. 3. There is a class for all unilateral one pass patterns employing symmetric crypto techniques, that is for which the authenticating party and the party requesting authentication share a common secret key which is used in the crypto protocol. Similarly, there is a class for all unilateral one pass patterns employing asymmetric crypto techniques, and a class for all patterns for which the unilateral one pass authentication algorithm employs a Hash function.

The patterns are then further specialized with respect to the authentication technique, or cryptographic protocol and algorithm(s) to be applied, e.g., for the unilateral two-pass authentication pattern, the HTTP digest authentication protocol with the MD5 hash algorithm may be applied [14]. By doing this, we separate out the choices that

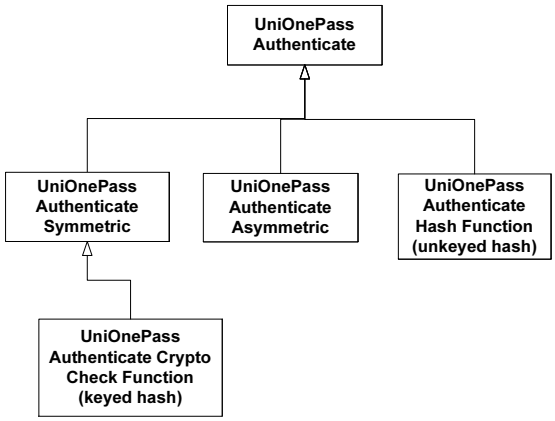

Figure 3: Unilateral one pass authentication patterns

must be made by the developer, and pinpoint each of the levels of specialization for awareness. This is because it is not enough to choose a general model and apply just any technique or protocol and assume that required level of security is achieved. By security level we mean the strength of authentication required to provide the required protection against misuse. There are altogether too many examples illustrating that depending on choices at each of these layers, the actual implementation can be flawed.

One example of this is the Microsoft challenge/reply handshake protocol, used in Microsoft's Point-to-Point Tunnelling Protocol (PPTP). In this example, a design flaw in the protocol and a choice of a weak password hashing algorithm both contributed to the reported weakness of the authentication implementation [47]. Additionally, there were other flaws in the implementation itself. It is because flaws may be introduced at different stages in authentication design and implementation that we have chosen to classify patterns separating stages of specialization. These stages are as follows: First, a general pattern is chosen from the classification in Fig. 2. Then, the pattern is specialized according to technique, e.g., if crypto is to be employed, then a choice must be made between symmetric or asymmetric keying, and then a protocol must be chosen along with algorithms or functions required by the protocol. If desired, an original protocol and algorithm may be designed for the application and specified during the design process. This will allow the developer to analyze authentication at each stage of specialization of the models, so that flaws and weaknesses may be discovered and corrected. For a more detailed discussion regarding choice of authentication pattern see Appendix C.1.1.

It is important to distinguish between weak versus strong authentication, and weaknesses and errors that arise simply due to implementation errors. The strength of the authentication pattern can be tuned with respect to the combination of the protocol, the algorithm and the key-length. However, errors in implementation can significantly weaken the authentication mechanism delivered. Assurance techniques such as e.g. use of the Common Criteria [21] may help in the latter.

3.2 Authorization patterns

In order to describe any authorization pattern, it is important to recognize that any authorization pattern requires that authentication has been performed before any au-

thorizations may be granted. Authentication and authorization patterns are combined to describe how access rights are granted and are thus essential to access control. Additionally, an access control model is required for access rights administration.

There are two basic authentication and authorization architectures [12]:

User Pull: Authentication is performed by an access server, which also issues authorizations to the user. The user then presents authorizations directly to the service.

Server Pull: The service centralizes information about user entity authorizations on an access server. The service authenticates the user. When the user attempts to access the service, the service queries the access server to determine whether the user is authorized.

These architectures provide a means for handling authorizations in a centralized manner. For the full classification of these architectures as patterns, see Appendix D.

4 Specification of AA-patterns

4.1 Specification of two-party authentication patterns

4.1.1 Using UML 2.0 collaborations

A UML 2.0 collaboration diagram for the generic two party authentication pattern is given in Figure 4. The collaboration diagram shows that the `authenticatee` role cooperates with the `authenticator` role.

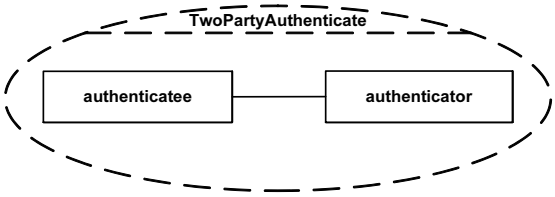

Figure 4: Collaboration diagram for the two party authentication pattern

A specialization of this pattern for unilateral two pass authentication is shown in Fig. 5. Using this specialization, an agent is able to authenticate another agent using a challenge response sequence in two passes. This view shows the goal for the collaboration, expressed in OCL.

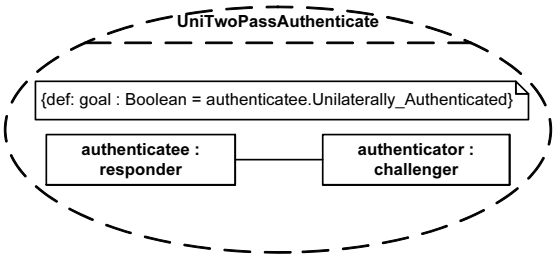

Figure 5: Unilateral two pass authentication

A detailed view of a specialization of this pattern for unilateral two pass authentication is shown in Fig. 6. This view expresses more completely the properties that the system components (such as agents) must have in order to successfully participate in the pattern. Any instance playing the `authenticatee` role must possess the properties specified by `responder` and any instance playing the `authenticator` role must possess the properties specified by `challenger`. The instance playing the `authenticatee` role must possess a secret, and the instance playing the `authenticator` role must possess knowledge that is mathematically related to the secret. The instance playing the `authenticator` role must be able to generate a `challenge`, which is sent to the instance playing the `authenticatee` role, and validate the `response`. Similarly, the instance playing the `authenticatee` role must be able to generate a `response` to the `challenge`. The constraints (on the properties that the instances playing the roles must possess) are declared as invariants and pre-conditions using the object constraint language (OCL).

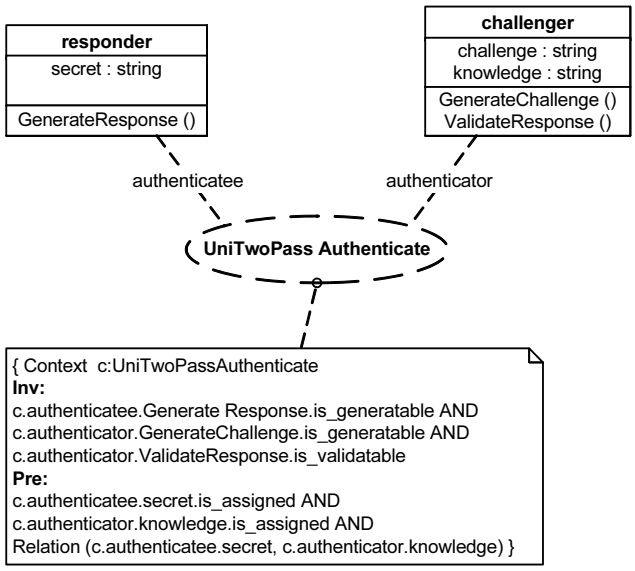

Figure 6: UML 2.0 Collaboration diagram for unilateral two-pass authentication, detailed view

Three invariants are declared: The first and third invariants are used to check that the instance playing the `authenticator` role is deployed on a part of the system (terminal/node) with the required processing and computing capacity required to generate the challenge and to validate the response. Similarly, the second invariant is used to check that the instance playing the `authenticatee` role is deployed on a part of the system (terminal/node) with the required processing and computing capacity required to generate the response. The reason for declaring these invariants is to ensure that the protocol and algorithm chosen are not too processor intensive for the parts on which they are deployed so that the authentication protocol can run whenever the collaboration is instantiated. The motivation for this is to ensure that service requirements regarding accessibility [37] are fulfilled when this authentication pattern is composed with service components/parts.

The two pre-conditions check that `secret` and `knowledge`, respectively, are

assigned before the collaboration can instantiate. The third pre-condition checks that there is a mathematical relationship between `secret` and `knowledge`. This means that a check can be performed to ensure that there is a pre-existing mathematical relationship between `secret` and `knowledge` as required by the authentication pattern to be deployed. The OCL pre-conditions can be used to perform a boolean check to confirm that the a priori conditions for the authentication protocol are fulfilled. This formalization of the mathematical relationship between `secret` and `knowledge` has been chosen in order to be general enough to allow for alternative crypto protocols to be specified at later stages in development. Note that if symmetric keying is used, then `secret` = `knowledge`.

4.1.2 Using semantic interfaces

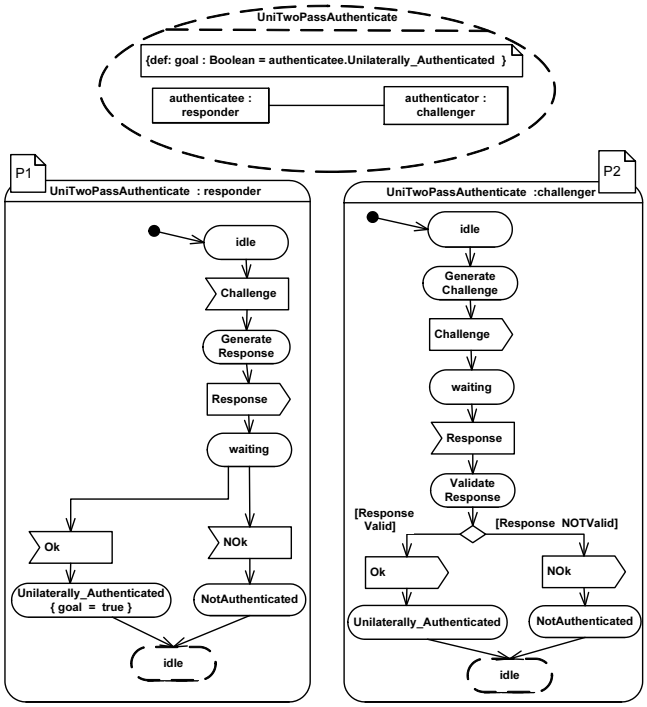

Figure 7: UML 2.0 collaboration and semantic interfaces for the unilateral two pass authentication pattern

A semantic connector is defined as an elementary collaboration with consistently defined pair of semantic interfaces and service goals [45]. The semantic interfaces may be modelled using two state transition diagrams defining the interface behavior for each of the roles involved in the collaboration and possible expressions stating the goals of the collaboration. In [46] it is described how semantic interfaces can be defined based on role modeling and simple goal expressions. The focus is on checking the compatibility of different service components involved in the provisioning of a service. Definition of semantic interfaces allows us to validate the interface behavior rather than validating the complete component behavior. Semantic interfaces facilitate validation of both safety and liveness properties. It is pointed out in [46] that UML 2.0 protocol

statemachines are not sufficient, and the authors propose a specification of a form of UML 2.0 state machines for two way interface behavior as shown in Fig. 7.

In this figure, the UML 2.0 collaboration for unilateral two-pass authentication pattern is shown with two role state machines that define the visible behavior of the two roles participating in the pattern.

In addition to syntactical interfaces, semantic interfaces define the visible interface behavior and goals of the collaboration. In this case, the semantic interface defines the interface behavior and goals of the `authenticatee` and `authenticator` roles.

Semantic interfaces in particular are projections of behavior on an interface and are characterized by:

- one action per transition
- spontaneous output
- visibility of variables and goals

4.2 Specification of n-party authentication and authorization patterns

We model the User Pull authentication and authorization services as a UML 2.0 collaboration that defines three collaborating participants that interact to implement the user pull authentication and authorization behavior: these are the `User`, `Access Server`, and `Service Access Filter` roles. Application of certain AA-patterns to the User Pull services is represented by three collaboration uses as illustrated in Fig. 8:

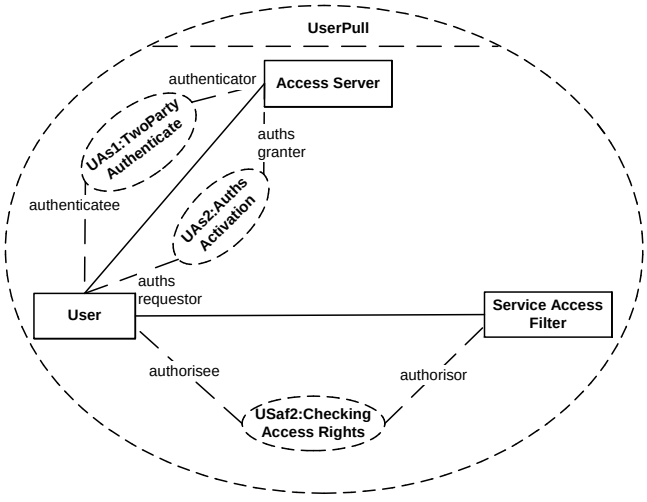

Figure 8: User Pull patterns

`TwoPartyAuthenticate:` This pattern, which we have modelled as a UML 2.0 collaboration in Fig. 4 and specialized for unilateral two pass authentication in Fig. 5 and Fig. 6, is shown in Fig. 8 bound to the `User` and `Access Server` roles. Here, the `authenticatee` role is bound to the `User` role, and the `authorisor`

role is bound to the Access Server role. For the instantiation of this pattern, it is expected that an appropriate two party authentication pattern is chosen and applied as described in Sect. 3.

Auths Activation: This pattern consists of a request by the instance playing the authsrequestor role for authorizations to be activated and sent to the instance playing the authsrequestor role. The authorizations govern which services the user is allowed to access. The way in which the authorizations are activated depends on the access control model that is used. This pattern is invoked after the collaboration TwoPartyAuthenticate has reached its goal of e.g., unilaterally authenticating the authenticatee. In the User Pull collaboration, Auths Activation is shown bound to the User and Access Server roles.

CheckingAccessRights: This pattern is invoked whenever the instance playing the User role requests access to a service. The instance playing the authoriser role then checks the authorizations to establish whether the instance playing the User role shall be granted access to the service. In the User Pull collaboration, Checking Access Rights is shown bound to the User and Service Access Filter roles.

Although not shown in the authentication and authorization patterns presented above, an access control model is needed to administer access rights (permissions) and enforce access control policies.

Several models for access control have evolved such as discretionary access control (DAC), mandatory access control (MAC), and others [12]. A detailed overview of different access control models is given in [56]. Role-Based Access Control (RBAC) has emerged as a scalable alternative, and has been the focus area for recent research on access control resulting in numerous model variants. In this report, we assume that a RBAC model is used with the AA-patterns.

RBAC-role activation rules, and authorization rules are administered by the RBAC infrastructure, and distributed to the AA-patterns and services. Therefore, there must be an interface from the Access Server towards an RBAC infrastructure. For an RBAC model and an approach to modelling RBAC policies using UML, see [35]. Access control policies are enforced based on RBAC-role activation rules and authorization rules. RBAC-activation rules are used to manage and activate RBAC-roles acquired by the agent. For example, a service role may or may not be allowed to be played by an agent depending on the RBAC-roles acquired by the agent. See Appendix D for for a more detailed discussion.

5 AA-patterns and policy

In [27], a policy is defined as information which can be used to modify the behavior of a system. This definition of policy covers as such role-binding constraints as well as user preferences, but also constraints on the triggering of behavior between components.

During service execution, dynamic role-binding provides a means for governing service execution as outlined in [7], using a policy-driven approach to control invocation of service roles. Our classification and approach to specification of policies is also motivated by [15] and by [29].

In our policy driven approach to composing AA-patterns with services we are concerned with defining selective mechanisms for enabling the joint behavior of objects rather than one object individually. As such policies should make it possible to provide information on sequencing of collaborative behavior as well as the triggering of collab-

orative behavior when policy constraints are fulfilled. It should be possible to provide information on the ordering of service goals as well as the relationship between collaboration uses composed to provide services. One way of doing this is expressing a composition policy as a UML2.0 dependency between two collaboration uses involved in a composed service as illustrated in Fig. 14.

Our notion of a role-binding policy specifies requirements/objectives specifically for the instance playing a certain role in the collaboration. This includes e.g. constraints the role imposes on any agent it may be bound to as well as conditions an agent may pose regarding which roles may be bound to that agent depending on agent states and preferences. Role-binding policies typically consist of context dependent constraints. In the context of authentication patterns, a collaboration policy is as such a requirement/objective for the collaborative behavior of the authentication pattern as a whole, whereas the role-binding polices are defined specifically for each of the two collaboration roles, `authenticatee` and `authenticator`.

Role-binding policies associated with a role may consist of:

- Role requirements, e.g. on which properties the instance (agent) playing the role must have in order to successfully participate in the collaboration. For example, for the unilateral two pass authentication pattern shown in Fig. 6, any instance playing the `authenticatee` role must possess a `secret`.
- Agent requirements, which may specify constraints on what the agent playing the role is allowed to do or which agents are allowed to play the role, e.g., only a `UserAgent` is allowed to play the `authenticatee` role. The constraint may specify requirements that the agent must satisfy in order to play the role, e.g., in order for the collaboration to be successful with respect to service availability requirements.
- Deployment requirements, e.g., requirements for the platform that the role is deployed on in order for the collaboration to be successful with respect to service availability requirements. For example, an instance playing the `authenticatee` role must be able to generate a `response`. This means that agent playing the role must be deployed on a part of the physical system with the required capacity available.

A collaboration role participating in an AA-pattern may have requirements on what the agent must be able to support in order to play a role. We therefore need to determine that the agent has the properties/characteristics required in order to play the role, such as support for a specific algorithm. If it is determined that the algorithm to be used is not supported, it may also be possible to download this (as a sort of extension to the role play) to the agent allowing for the role to be played anyway.

A role-binding policy held by an agent defines conditions and constraints on which roles can be played by the agent and defines rules in terms of:

- Pre-conditions for invoking a role such as conditions on the other agent involved in the collaboration or conditions on which roles shall have been performed (e.g. AA-roles).
- Preferences of the agent, such as types or multiplicities of roles that can be bound to the agent.
- Deployment conditions. This provides e.g., Information about the resources available. This may include information about the type of terminal/node/user

equipment that the agent is deployed on, e.g., the terminal is a 3G telephone with a smartcard, information about the operating System/ or software supported, and other contextual parameters. Essentially, providing information about which service availability constraints can be met by the platform the agent is deployed on and which influence whether or not a role can be played by an agent.

For example, the policy held by a user agent `A` may state that user agent `A` is only allowed to participate in a `voicecall` service, playing the `callee` role if the user agent playing the `caller` role has been authenticated, authorized, and identified, and the invitation is received between 6 PM and 11 PM.

5.1 Specifying policies

Most of the work in the literature on defining policies focuses on organizational policies e.g. RBAC policies and Role Based Management (RBM) policies in which a role is an organizational concept representing the specification of the behavior associated with a particular position in the organizational context [26]. Although policies for governing service execution are addressed in [7], specification of rules for defining such policies is not addressed. An architecture for policy definition and call control policies is given in [36] and provides some high level ideas for defining policies for use in enhancing and controlling features in the context of call control in telecommunication services. In this section, we refine some of these ideas and we provide our approach to defining policies.

Definition: A policy is a rule of the following form: If *condition* C and *trigger* T then *action* A and *goal* B.

- *The condition part* defines constraints on its applicability. The constraints are predicates which restrict role behavior in service composition. We may specify constraints as invariants, and pre and post conditions in OCL, or more specifically, in Ponder [9].

- *The trigger part* describes when the policy should be applied. The trigger is the event that e.g. invokes the execution of a collaboration subject to the constraints stated in the condition part. The trigger part of a policy for governing service invocation is important for achieving dynamic linking in service composition. The trigger is specified as a message in UML 2.0, e.g., a signal or call.

- *The action part* defines what is to be done when the trigger event has been sent given that the constraints stated in the condition part hold. Examples of actions are: *bind role r to agent A*, and *execute collaboration C*.

- *The goal part* defines what is the desired result when the policy is applied. These goals may be specified as post-conditions in OCL.

Although a trigger part is not specified in policy rules in general, e.g. in [15], the trigger part is essential for applying a policy approach to service engineering. For a role-binding policy, the trigger establishes when the policy applies, e.g., when the role request message is sent. Specifying composition policies allows us to make relationships between collaboration uses explicit as well as providing a means for sequencing service goals. e.g., a composition policy may state that the goal `unilaterally_authenticated` must be achieved before the goal `auths_activated` can be achieved.

5.2 Applying policies to service composition

The following outlines how policy is applied in our modeling:

1. The role-binding and composition policies are specified using e.g., OCL. Policy conditions are stated as invariants, and pre- and post conditions. Triggers and policy actions are stated as UML operations, and goals as post-conditions. For example, if the invariants and pre-conditions stated in the role-binding policy are satisfied, then the instance can play the role. Another example is use of composition policies to demonstrate dynamic linking of collaboration uses. The composition policy is declared as a UML 2.0 dependency of type $\ll policy \gg$. For example, the composition policy may declare that a pre-condition for the execution of one collaboration may be that the instantiation of another collaboration has reached a certain goal. These policies are specified at design time.

2. At design time static checks are performed on e.g., the projection from an actor's state machine to the semantic interface. Checks are performed on role compliance. This includes checking that the actor satisfies the conditions and properties given in the role-binding policy. This implies that the actor is typed with the interface.

3. At run-time, policy controls are performed on the interfaces, dynamically. At run-time it is enough to check that both instances are of the types that are required on the semantic connector. Whether collaboration policy is satisfied is checked, as well as checking e.g., whether access control policy rules are satisfied.

4. Access Control Policy enforcement is performed dynamically in an instantiation of the `Checking Access Rights` collaboration by the instance playing the `authorisor` role.

We have found that it is useful to declare the role binding policies in the semantic interfaces for use in validation that the security properties are preserved in composition of the pattern with services.

6 Composing AA-patterns and services statically

AA-patterns behavior may be invoked in two different situations:

When creating a new session, by performing a role request and performing dynamic role binding. This requires general mechanisms to ensure that the role is invoked only if authentication and authorization policies are satisfied. If role `r` is requested, and a policy specifies that authentication and authorization is performed first, then the necessary AA-behavior must be performed first and a desired goal must be reached before the service is invoked. In this case an AA-goal is a precondition for the service invocation.

During session behavior, this is required when the session and its roles contains features or accesses objects that demand fine-grained, dynamic authentication and/or authorization. This case is trickier because it requires a tighter integration of service behavior and AA-behavior. In our work, we model this using service access filters, and policies, e.g. restricting role behavior. This entails adding screening behavior that filters out unauthorized operations. It also requires that it may be possible to force termination of a session if authorizations are no longer valid. We have currently modeled this

as an `Interrupt` collaboration. Another approach is to invoke a restricted role behavior only capable of doing authorized operations. Applying the appropriate role-based access control model for issuing authorizations, and checking authorizations upon accessing a particular service or object makes such fine-grained, dynamic authorization possible.

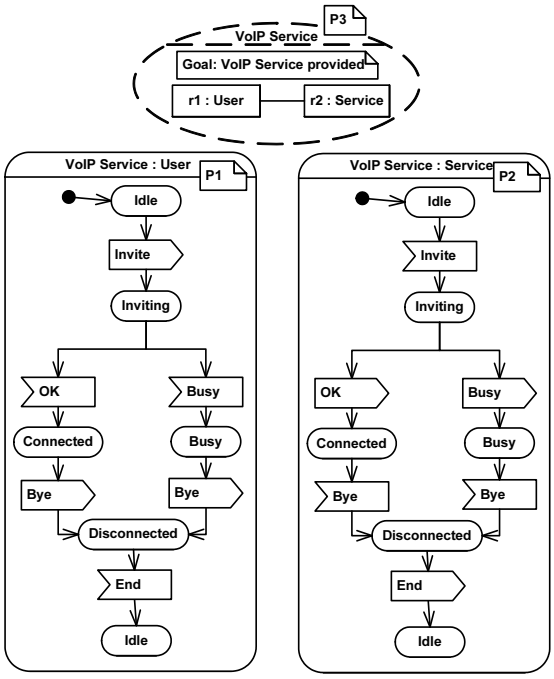

Figure 9: VoIP service defined as a semantic interface

Let us assume a voice over IP (VoIP) service, `VoIP Service`, defined as a semantic interface with roles `r1` and `r2` as shown in Fig. 9. We model the view showing the user to VoIP service provider only, to keep the example simple. Further assume that agent `A` requests a session of `VoIP Service`, and role `r2` from agent `B`.

The collaboration `VoIP Service` may have a collaboration policy `P3` specifying that the agents playing `r1` and `r2`, in our case agents `A` and `B`, shall be different agents. The agents may specify conditions that govern which roles can be played by the agent. Agent `B` may, for instance, specify that a precondition for invoking `r2` is that agent `A` is authenticated and authorized e.g. applying `Userpull`. Similarly, agent `A` may specify that a precondition for invoking `r1` is that agent `B` is authenticated and authorized. It is natural to express these conditions as part of the role-binding policies, using OCL.

If the AA-properties have not been established yet then, it is necessary to invoke AA-services resulting in the desired AA-properties before invoking `VoIP Service`. In the most general case agent A and agent B must negotiate and agree on the AA-patterns to apply. In many cases agent B may select the patterns and return the decision to agent `A`. Then the AA-services are performed and only if successful, is the requested `VoIP Service` invoked. In Fig. 10 we illustrate the mapping of the `VoIP Service` collaboration to agents in the system, however, it should be noted that this is not syntactically legal in UML 2.0 [32], although this would be useful.

In order to demonstrate composition of VoIP Service with AA-patterns, we decom-

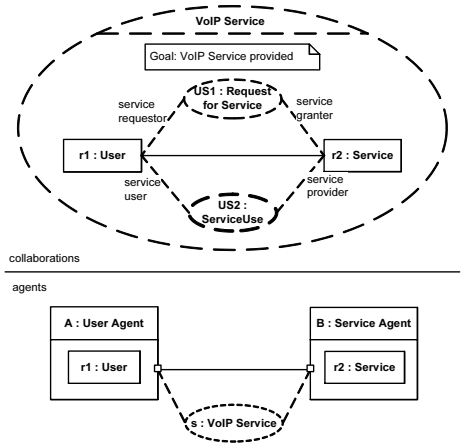

Figure 10: VoIP Service: binding roles to agents in service composition

pose `VoIP Service` as shown in Fig. 10. The collaboration Use `Request for Service` represents the initial request for use of the VoIP service by the instance playing the `User` role. There are several alternatives to determine what is a result of this initial request. One option, is that a service manager is implemented in the system, which in response to the request from the user determines that authentication and authorization is required for access to the service. A set of AA-patterns is then selected for composition with the service. Another alternative is that the instance playing the `Service` role determines which AA-patterns are needed and that instances of Service Access Filter are required to perform authentication and authorization, and if successful, then the requested `VoIP Service` is invoked.

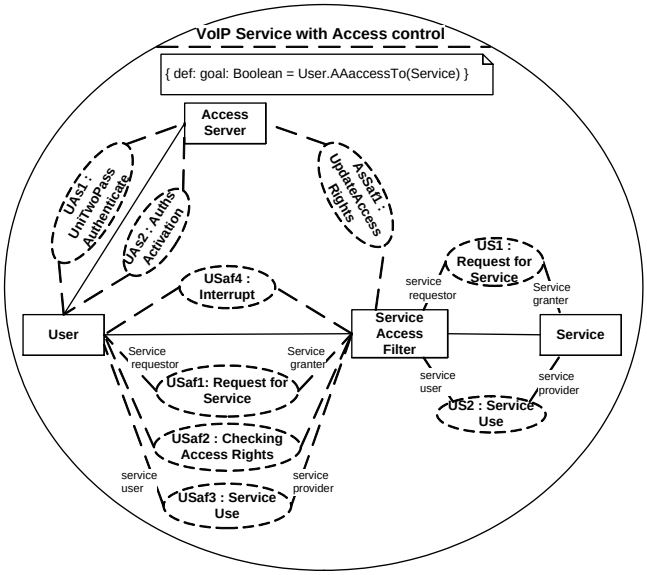

Figure 11: VoIP Service composed with User Pull patterns

As this report focuses primarily on modelling techniques/alternatives for enabling static composition of AA-patterns and services at design time, we do not discuss the dynamic linking that occurs from Fig. 10 to Fig. 11. We assume, therefore that the decision to compose the AA-patterns with the VoIP Service as shown in Fig. 11, the collaboration `VoIP Service with Access control`, has been made. We now discuss the different modelling techniques/alternatives for achieving static composition.

In Fig. 11 we demonstrate static composition of `VoIP Service` with the User Pull authentication and authorization patterns. This involves re-use of the two collaborations: `Request for Service` and `ServiceUse`. The re-use of these two patterns is needed in order to enable the instance playing the `ServiceAccessFilter` role to act as a proxy between the instance playing the `User` role, and the instance playing the `Service` role. This enables the instance playing the `Service` role to require authentication and authorization before allowing a user to access the service. The `VoIP Service` session may require additional, fine grained authentication, and authorization checks, however, and this calls for screening or other mechanisms during service execution, unless it is possible to constrain the service that is invoked to what is permitted. The instance playing the `Service` role, may require that these additional, fine grained authentication, and authorization checks are performed by the instance playing the `ServiceAccessFilter` role. We model these as the following collaboration uses: `UpdateAccessRights`, for updating the status of the user authorizations, and `Interrupt`, for terminating a service session if user authorizations are no longer valid.

AA-pattern collaborations describe reusable elements. During instantiation of a collaboration, various checks are needed to ensure that the participating agents can satisfy requirements, conditions and properties, stated in policies.

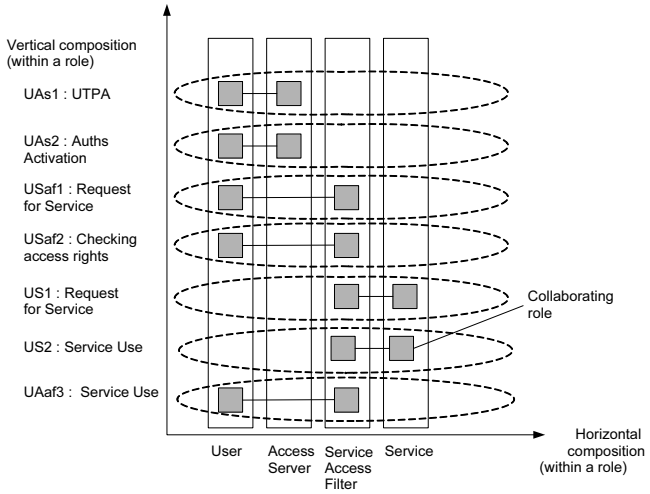

Figure 12: Diagram showing composition of collaboration uses, ordered in top down sequence

Role-binding policies are used to check the compatibility of the role with the agent playing the role. When binding roles, the semantic interface between two roles is also bound, that is, the roles must also be compatible on a semantic interface with each

other. Work on validating the compatibility of roles and consistency checking to ensure the correctness of roles has been done by [13], [44], and [10].

Fig. 11 gives a graphical overview and provides a decomposition into interfaces that are quite modular and reusable. However, the overall coordination (referred to as choreography in the SOA context) is not evident. In addition to providing information about the static structure, we also need to provide information about the ordering of the associated behavior. Fig. 12 shows how roles in the different collaborations are composed in and how these are ordered in a successful service execution.

As explained, above Section 4, for each of the two party collaborations we model the behavior associated with the collaboration using semantic interfaces and goals. In addition, a UML 2.0 interactions diagram corresponding to the semantic interface may be designed for each collaboration. These can then be referred to in a UML 2.0 interactions overview diagram such as in Fig.13.

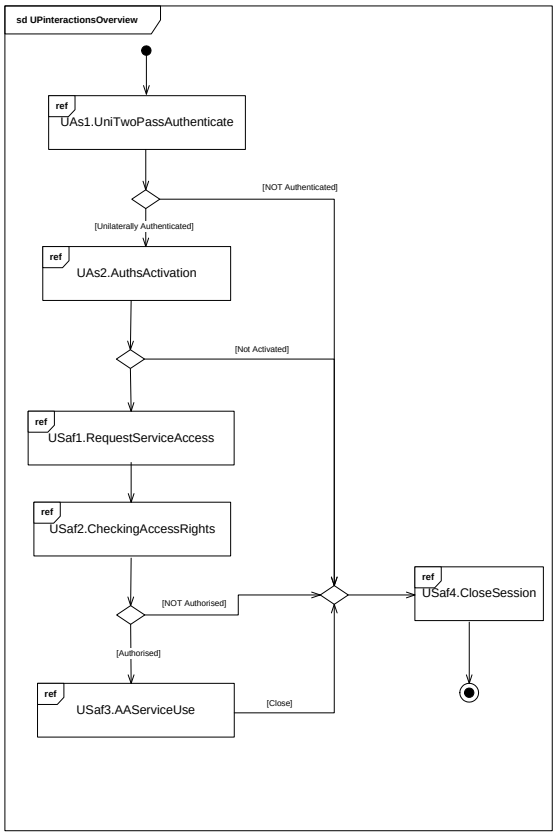

Figure 13: Interactions overview diagram for composing sequence diagrams

There are several alternatives for modelling the sequencing of the behavior associated with the collaboration uses shown. One alternative is use of an interactions overview diagram as shown in Fig. 13. Such interaction overview diagrams are not entirely suitable for expressing interrupting and disabling such as the termination of a user session if authorizations are no longer valid. To model such dynamic exceptions, a UML 2.0 activity diagram may be useful, modelling the dynamic exception using an interruptible activity region [32].

Modelling the `Interrupt` collaboration behavior in service composition is not easy, as it constitutes an exception behavior. Indeed, the `Interrupt` collaboration is an example of a forced feature interaction. Halvorsen and Haugen have presented a method for handling exception in sequence diagrams in [16].

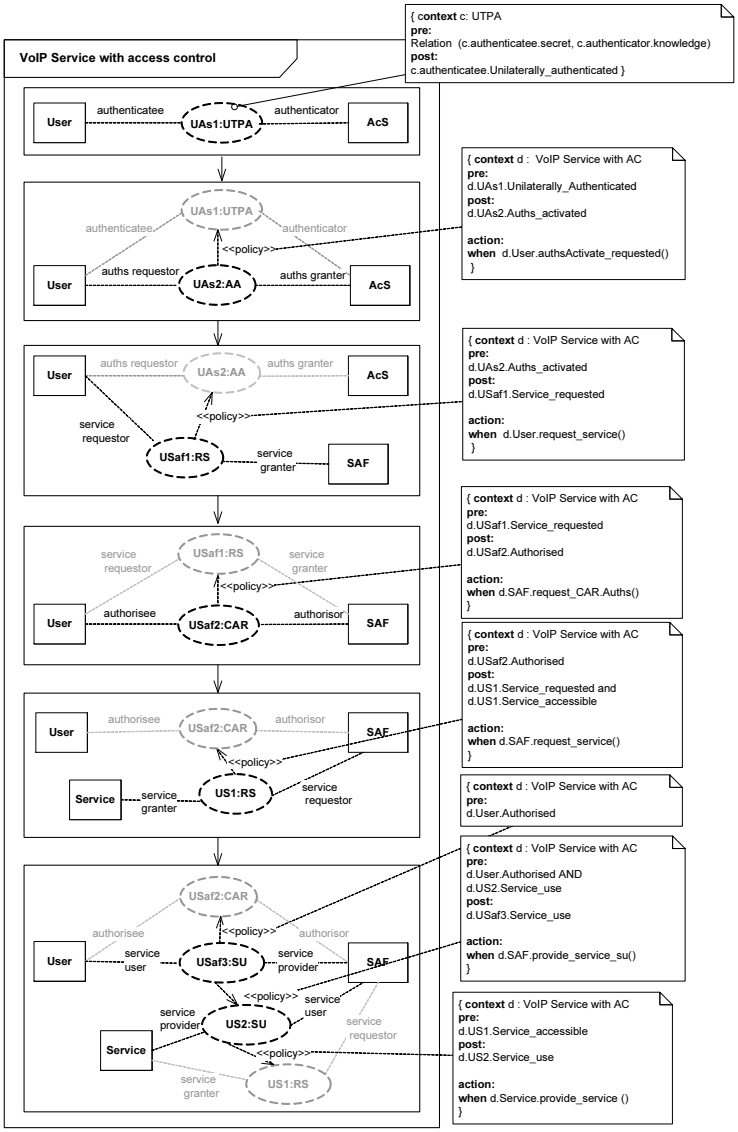

Figure 14: Goal sequences with policies as UML 2.0 dependencies

A goal sequence [43] provides supplementary information to a collaboration diagram. While a collaboration such as given in Fig. 11 provides static structural information about the roles and collaboration uses involved in a composition of re-usable units such as AA-patterns and services, a goal sequence provides additional information about the ordering of dynamic behavior associated with the collaboration uses. Fig. 14 models the positive sequence of behavior associated with the collaborations in

order to achieve authenticated and authorized access to a service.

We have extended the idea of a goal sequence given in [43] to include modelling composition policies as UML 2.0 dependencies with keyword $<<$`policy` $>>$ as illustrated in Fig. 14. This allows us to express constraints on the ordering of the behavior associated with collaboration uses and may also allow us to express policies governing dynamic interupt exceptions. The composition policies, modelled as UML 2.0 dependencies with keyword $<<$`policy` $>>$ allow us to specify conditions that must be true in order for the behavior associated with a collaboration use to execute. Each instance of $<<$`policy` $>>$ is annotated with the policy specified in OCL.

The policies declared provide additional information on conditions required for the behavior to run correctly and according to availability requirements. The post-conditions declare goals that have been defined at design time in the semantic interfaces for the collaborations involved. For example, the goal `unilaterally_authenticated` is declared in the semanic intefrace for `UTPA` in Fig. 16.

As shown in Fig. 14, a collaboration policy is declared using OCL for the first collaboration use in the sequence, the instance `UAs1` of `UTPA`. This collaboration policy is declared at design time, when the `UTPA` collaboration is designed, along with UML 2.0 interactions and the semantic connector and its pair of semantic interfaces. The goal for the collaboration, `c.authenticatee.Unilaterally_authenticated`, is also declared in the collaboration policy as an OCL post-condition. The reaching of this goal, becomes a pre-condition in the composition policy declaring when the behavior associated with the instance `UAs2` of `AA` can execute.

We prefer to model the composition policies as UML 2.0 dependencies in a goal sequence as apposed to declaring such policy dependencies in a UML2.0 collaboration overview such as the overview shown in Fig. 11. This is because, the dependencies would cross over several collaboration uses, and often cross each other, making the result very difficult to read and understand. By using a goal sequence instead, the policies can be expressed clearly, sequentially, and dynamically.

6.1 Steps for composing AA-patterns with services

We sum up our approach to composing AA-patterns at design time in the following steps:

1. Determine which AA-patterns should be applied. In this step, it is determined based on service availability requirements, which set of AA-patterns will be applied. This involves deciding whether AA-behavior should be applied separately for each service in parallel, or whether some form of centralized authentication and authorizations can be used, requiring that either the `UserPull` patterns or `ServerPull` patterns should be applied. The decision to apply `UserPull` or `ServerPull` involves deciding whether user authorizations will be stored on a centralized access server, and presented by the access server to the service, or whether authorizations will be distributed to the user and presented by the user to the service. Regarding choice of authentication pattern to apply, we discuss this in more detail in C.1.1

2. Decide whether sequential invocation at the beginning of a session only is sufficient, or whether more fine grained control during session behavior is also required.

3. Once the set of patterns to apply has been chosen, specifications/models for each of the AA-patterns and the service to be composed are designed. These are: UML 2.0 collaborations annotated with goals and a collaboration policy, semantic interfaces annotated with role binding policies for each of the two participating roles, and UML 2.0 interations. Declaring role-binding policies in the semantic interface for each of the two collaborating roles involved in the semantic connector will enable us to validate that the required conditions and requirements have been fulfilled when composing the pattern with other AA-patterns and services. The semantic connector may also be annotated with the collaboration policy and goals.

4. Specification of the collaboration showing composition of AA-patterns with the service (annotated with the collaboration policies. To supplement this collaboration overview diagram, a goal sequence diagram is also provided, e.g. as shown in Fig. 14.

5. Consistency checking of the model in the previous step using semantic interfaces. Consistency checks related to goals, and to role binding policies. In this step we will also evaluate whether or not availability properties are preserved under composition.

While these steps address static composition at design time, it is also possible that agents representing users in the system, negotiate on behalf of end-users, service providers and system resources to achieve dynamic composition at run time, as we discuss in the following section.

7 Dynamic role-binding using semantic interfaces

We define the semantic interfaces (SI) separately and validate (model check) each SI type separately to ensure safety and liveness properties. In this sense, a semantic interface is a type that may be used at design time to ensure the correctness of (static) associations and at runtime (as meta information) to ensure the correctness of (dynamic) links.

Role binding policies declare requirements for the classes and instances a role may be bound to. Actor/role types are then designed for the runtime system and are model checked against the SI to validate that the interface behavior required by the collaboration (e.g., UTPA) is satisfied.

In the meta data for the runtime system, this information is stored in files in the database as part of the management system, and forms part of the data model for the runtime system. In the meta data, we know that the instance can play the authenticatee role of type responder in an instantiation of the UTPA collaboration and be able to satisfy requirements regarding strength of authentication provided, and response times involved in the exchange.

At runtime, dynamic role-binding is performed using the actor and SI type information to ensure compatibility of dynamic links thereby guaranteeing that the links satisfy the properties of the SI.

7.1 Example

The semantic connector for a specialization of the unilateral two pass authentication pattern is given in Fig. 16 along with the `UserAgent` and `Service Agent` which

represent user and service domain entities and resources, respectively. In order for an instance of `responder` to be bound to the `UserAgent`, the role-binding policy `P1` must be satisfied, similarly, for binding and instance of `challenger` to the `Service Agent`. The role-binding policies have been specified in OCL for the roles involved in the two party authentication pattern. For example, the OCL Boolean constraint `is_generatable` is declared to address performance aspects of the authentication exchange. The aim is to ensure that the system resource on which the role/agent is deployed is able to perform the operations involved in the authentication exchange within QoS requirements.

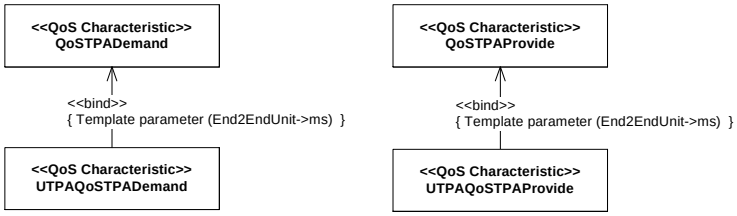

Figure 15: UTPA QoS class definition

The OCL Boolean constraint `is_generatable` has been defined using [33], and `is_generatable` evaluates to true means that the *required QoS* demanded by the role in order to satisfy accessibility constraints is met by the *offered QoS* of the resource in the deployment model. In order to represent the quality values we need to define `is_generatable` for use in dynamic role-binding, we define a simplified quality model, as shown in Fig. 15 based on the Quality Model given in Annex B of [33]. In this case, we resolve all temporal units with the unit milliseconds (ms). This simple model may be expanded and refined with additional characteristics.

The OCL constraint `is_generatable` is a Boolean check that is defined is OCL as follows:

{ **Context** UTPA
if UTPAQoSTPAProvide $\leq$ UTPAQoSTPADemand
then self.authenticatee.generateMD5response.is_generatable = true
else self.authenticatee.generateMD5response.is_generatable = false
endif
}

The required QoS defines the maximum allowed time to generate the MD5 response, and is specified on the SI type annotated to the statechart for the `responder` role. In this case, the required QoS is the worst case for generating the MD5 response is 10 ms. Similarly, the deployment model for the agents provides information about the offered QoS of the resources. The offered QoS by the resource is 10 ms or better for the agent that is to be validated against the required QoS. For this case, as shown in Fig. 16, `is_generatable` evaluates to true.

Support for java-based role-binding, and collaboration policies has been implemented in ServiceFrame [5]. Services can be specified by both end-users and service providers to handle availability properties. Extensions of ServiceFrame for validation interface behavior by checking consistency are also being investigate by the students. Extensions for modelling collaborations and deriving interface behavior associated with these have also been implemented. Work is ongoing regarding consistency of service roles using semantic interfaces.

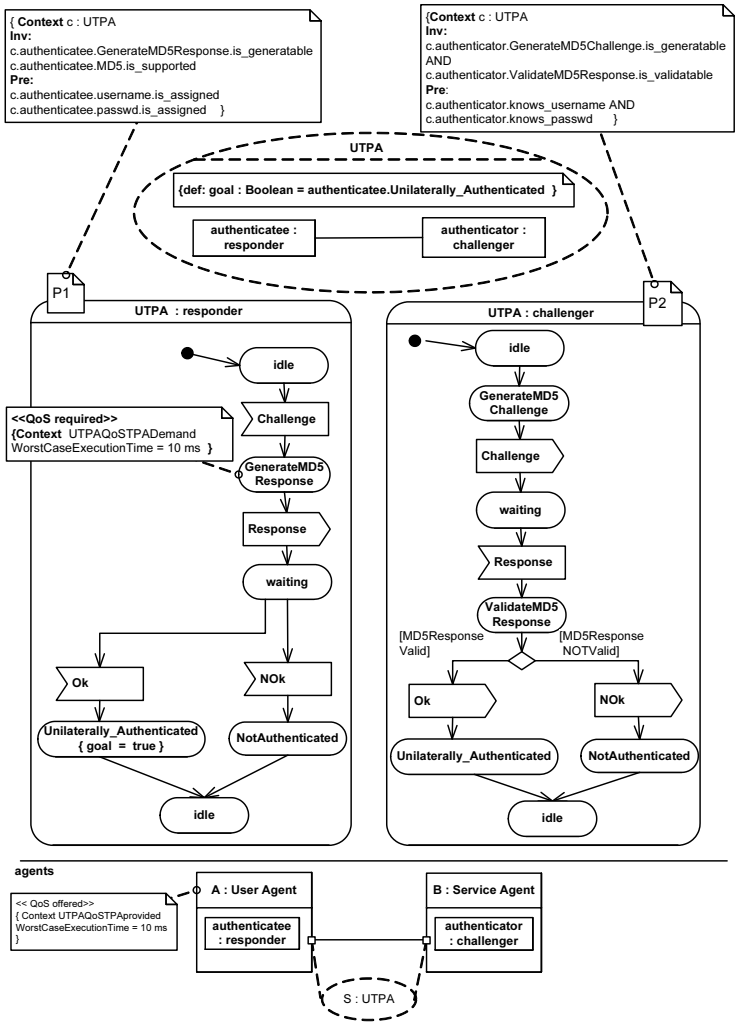

Figure 16: Semantic connector for a specialization of the unilateral two pass authentication pattern

8 Related work

Yoder and Barcalow [58] were the first to apply design patterns to the security domain presenting the Single Access Point Pattern in [58]. In [8], patterns for authorization and access control are addressed. Brown, Divietri, Villegas, and Fernandez have documented a high level design pattern for authentication of clients to a server [6]. Consistent with our approach, the pattern allows for the implementation of different authentication methods such as password-based, challenge response, or multiple challenge response. However, our approach to designing patterns allows for application of the authentication pattern to the peer-to-peer environment as well. Additionally, we provide a means to specify more details at later stages of development depending on the requirements of the authentication protocol and algorithm. In [11], Fernandez and Warrier provide an authorization pattern, integrated with a variant of the authentic-

ator pattern. This authorizer pattern is actually an application of Yoder and Barcalows single-point-of-check pattern [58], and is also an example of a server pull authentication and authorization architecture. Although these and other different authors have addressed authentication patterns and authorization patterns separately, we are not aware that a framework addressing authentication and authorization patterns exists. To our knowledge, application of such a framework to service composition is also a new approach.

9 Conclusion

We have presented a framework of authentication and authorization patterns together with a policy-driven approach to composing services and AA-patterns to restrict access to services to authorized users only. This involves specification of the AA-patterns using UML 2.0 collaborations and semantic interfaces annotated with policies specified using OCL. We have demonstrated that our framework can be applied to static and dynamic composition of services. Furthermore, we have demonstrated how the specifications may be annotated with role-binding policies, collaboration policies, and composition policies to enable us to validate that required conditions and availability properties hold when composing AA-patterns with services.

This policy-driven approach is useful for application to service composition because there are significant differences between different authentication techniques that must be modelled for use in service composition, depending on the service collaboration roles and service behavior involved as well as differences in the resources available in the deployment platform. This validates the need for a finer-grained classification of authentication patterns as discussed above in Sect. 3 and in Appendix C.

Acknowledgment

Thanks to Manfred Broy, Humberto Nicolas Castejón, Øystein Haugen, Frank Alexander Kraemer, Mass Soldal Lund, Birger Møller-Pedersen, Ragnhild Kobro Runde, Richard Sanders, Ina Schieferdecker, Ketil Stølen, and Thomas Weigert for commenting on earlier versions of this paper.

References

[1] G.-J. Ahn and M. E. Shin. Role-based authorization constraints specification using object constraint language. In *Proceedings of the 10th IEEE International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE 2001)*, pages 157–162. IEEE Computer Society, June 2001.

[2] C. Alexander. *The Timeless Way of Building*. Oxford University Press, 1979.

[3] W. A. Arbaugh, W. L. Fithen, and J. McHugh. Windows of vulnerability: A case study analysis. *IEEE Computer*, 33(12):52–59, December 2000.

[4] C. Boyd and A. Mathuria. *Protocols for Authentication and Key Establishment*. Series: Information Security and Cryptography. Springer-Verlag, 2003.

[5] Rolv Bræk, K E Husa, and G Melby. ServiceFrame: WhitePaper. August 30, 2006 [online] – URL : http://ikt.hia.no/teleservice/ServiceFrameWhitepaperv8.pdf, April 2002.

[6] F. L. Brown, J. Divietri Jr., G. D. Villegas, and E. D. Fernandez. The authenticator pattern. Proceedings of Pattern Language Programs (PLoP99), August 1999.

[7] H. N. Castejón and R. Bræk. Dynamic role binding in a service oriented architecture. In *Proceedings of the 2005 IFIP International Conference on Intelligence in Communication Systems (INTELLCOMM)*, volume 190. Springer-Verlag, October 2005.

[8] B.H.C. Cheng, S. Konrad, L. A. Cambell, and R. Wassermann. Using security patterns to model and analyze security requirements. In *Proceedings of the 2nd International Workshop on Requirements for High Assurance Systems (RHAS 03)*, September 2003.

[9] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. Ponder: A language for specifying security and management policies for distributed systems. Imperial College Research Report DoC 2000/1, Department of Computing, Imperial College of Science, Technology and Medicine, October 2000.

[10] Fritjof Boger Engelhardtsen and Andreas Prinz. Application of stuck-free conformance to service-role composition. Presented at the 5th Workshop on system analysis and modelling (SAM 2006), June 2006.

[11] E. D. Fernandez and R. Warrier. Remote authenticator/authorisor. Presented at the 10th Conference on Pattern Language Programs (PLoP2003), August 2003.

[12] D. F. Ferraiolo, D. R. Kuhn, and R. Chandramouli. *Role-Based Access Control*. Artech House, 2003.

[13] J. Floch and Rolv Bræk. A compositional approach to service validation. In *Proceedings of the 12th International SDL Forum (SDL 2005)*, pages 281–297. Springer-Verlag, June 2005.

[14] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. HTTP authentication: Basic and digest access authentication. RFC 2617, June 1999.

[15] Z. Fu, F. Wu, H. Huang, K K. Loh, F. Gong, I. Baldine, and C. Xu. IPsec/VPN security policy: Correctness, conflict detection, and resolution. In *Proceedings of the International Workshop on Policies for Distributed Systems and Networks*, pages 39–56. Springer-Verlag, 2001.

[16] O. Halvorsen and O. Haugen. Proposed notation for exception handling in UML 2.0 sequence diagrams. In *Proceeding of the Australian Software Engineering Conference (ASWEC 2006)*, pages 29–42. IEEE Computer Society, April 2006.

[17] International Standards Organization. *ISO 7498-2, Information Processing Systems – Interconnection Reference Model – Part 2: Security Architecture*, 1989.

[18] International Standards Organization. *ISO/IEC 9798-4, Information technology – Security techniques – Entity Authentication Part 4: Mechanisms using a cryptographic check function*, 1995.

[19] International Standards Organization. *ISO/IEC 9798-1, Information technology – Security techniques – Entity Authentication Part 1: General*, 1997.

[20] International Standards Organization. *ISO/IEC 9798-3, Information technology – Security techniques – Entity Authentication Part 3: Mechanisms using digital signature techniques*, 1998.

[21] International Standards Organization. *ISO/IEC 15408, Information technology – Security techniques – Evaluation criteria for IT security*, 1999.

[22] International Standards Organization. *ISO/IEC 9798-2, Information technology – Security techniques – Entity Authentication Part 2: Mechanisms using symmetric encipherment algorithms*, 1999.

[23] International Standards Organization. *ISO/IEC 9798-5, Information technology – Security techniques – Entity Authentication Part 5: Mechanisms using zero knowledge techniques*, 1999.

[24] International Standards Organization. *ISO/IEC 13335, Information technology – Security techniques – Guidelines for the management of IT security*, 2001.

[25] A. K. Lenstra and E. R. Verheul. Selecting cryptographic key sizes. In *Proceedings of the 3rd International Workshop on Practice and Theory in Public Key Cryptography*, pages 446–465. Springer-Verlag, January 2000.

[26] E. Lupu and M. Sloman. Reconciling role based management and role based access control. In *Proceedings of the 2nd ACM Workshop on Role Based Access Control*, pages 135–141. ACM Press, November 1997.

[27] E. C. Lupu and M. S. Sloman. Conflicts in policy-based distributed systems management. *IEEE Transactions on Software Engineering*, 25(6):852–869, November-December 1999.

[28] A. J. Menezes, P. C. van Oorchot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.

[29] J. D. Moffett and M. S. Sloman. Polcy conflict analysis in distributed systems management. *Journal of Organizational Computing*, 4(1):1–22, January 1994.

[30] R.M. Needham and M.D. Schroeder. Using encryption for authentication in large networks of computers. In *Communications of the ACM*, pages 993–999. ACM Press, December 1978.

[31] A. Niemi, J. Arkko, and V.Torvinen. HTTP digest authentication using authentication and key agreement (aka). RFC 3310, September 2002.

[32] Object Management Group. *UML 2.0 Superstructure Specification, formal/05-07-04*, 2006.

[33] Object Management Group. *UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms, formal/06-05-02*, 2006.

[34] J. Park, R. Sandhu, and G. Ahn. Role-based access control on the web. *ACM Transactions on Information and System Security*, 4(1):37–71, February 2001.

[35] I. Ray, Na Li, R France, and Dae-Kyoo Kim. Using UML to visualize role-based access control constraints. In *Proceedings of the Ninth ACM symposium on Access control models and technologies (SACMAT 2004)*, pages 115–124. ACM Press, June 2004.

[36] S. Reiff-Margeniec and K. J. Turner. A policy architecture for enhancing and controlling features. In *Proc. Feature Interactions in Telecommunication Networks VII*, pages 239–246. IOS Press, June 2003.

[37] J. E. Y. Rossebø, M. S. Lund, K. E. Husa, and A. Refsdal. A conceptual model for service availability. Research report 337, Department of Informatics, University of Oslo, June 2006.

[38] J. E. Y. Rossebø, M. S. Lund, K. E. Husa, and A. Refsdal. A conceptual model for service availability. *Quality of Protection: Security Measurements and Metrics*, 23, August 2006.

[39] J. E. Y. Rossebø, J. Ronan, and K. Walsh. Authentication issues in multi-service residential access networks. In *Proc. Seventh International Conference on Management of Multimedia Networks and Services (MMNS'2003)*, pages 381–395. Springer-Verlag, September 2003.

[40] Judith E Y Rossebø and Rolv Bræk. A policy-driven approach to dynamic composition of authentication and authorization patterns and services. *Journal of Computers (JCP)*, 1(8):13–26, December 2006.

[41] Judith E Y Rossebø and Rolv Bræk. Towards a framework of authentication and authorization patterns for ensuring availability in service composition. In *The First International Conference on Availability, Reliability and Security (ARES 2006)*, pages 206–215. IEEE Computer Society, 2006.

[42] The European Telecommunication Standardisation Institute (etsi) Security Experts Group (sage). September 20, 2005 [online] – URL : http://portal.org/sage/Summary.asp.

[43] R. Sanders, H. N. Castejón, F. Kraemer, and R. Bræk. Using UML 2.0 collaborations for compositional service specification. In *MoDELS 2005*, pages 460 – 475. Springer-Verlag, 2005.

[44] R. T. Sanders and Rolv Bræk. Modeling peer-to-peer service goals in UML. In *Proceedings of the 12nd IEEE International Conference on Software Engineering and Formal Methods (SEFM 2004)*, pages 144–153. IEEE Computer Society, September 2004.

[45] Richard Sanders. Collaborations, semantic interfaces and service goals: a way forward for service engineering. Ph.D. thesis NTNU, submitted, December 2006.

[46] R.T. Sanders, R. Bræk, G. von Bochmann, and D. Amyot. Service discovery and component reuse with semantic interfaces. In *Proceedings of the 12th International SDL Forum (SDL 2005)*, pages 85–102. Springer-Verlag, 2005.

[47] B. Schneier and D.W. Mudge. Cryptanalysis of microsoft's point-to-point tunnelling protocol (pptp). In *Proceedings of the 5th ACM Conference on Communications and Computer Security*, pages 132–141. ACM Press, November 1998.

[48] R. Shirey. *Internet Security Glossary. RFC 2828*. Network Working Group, 2000.

[49] Standards Australia. *AS/NZS 4360:1999, Risk Management*, 1999.

[50] Marc Stevens, Arjen Lenstra, and Benne de Weger. Target collisions for md5 and colliding x.509 certificates for different identities. Technical report, International Association for Cryptographic Research, October 2006.

[51] Third Generation Partnership Project, Technical Specification Group Services and Systems Aspects, 3GPP, TR 21.905 V 7.2.0 (2006-06). *Vocabulary for 3GPP Specifications (Release 7)*, 2006.

[52] Third Generation Partnership Project, Technical Specification Group Services and Systems Aspects, 3GPP, TS 33.203 V 7.3.0 (2006-09), 3G Security. *Access Security for IP-based Services (Release 7)*, 2006.

[53] Third Generation Partnership Project, Technical Specification Group Services and Systems Aspects, 3GPP, TS 35.205 V 6.0.0 (2004-12), 3G Security. *Specification of the MILENAGE Algorithm Set: An example algorithm set for the 3GPP authentication and key generation functions f1, f1*, f2, f3, f4, f5, and f5*; Document 1: General (Release 6)*, 2004.

[54] Third Generation Partnership Project, Technical Specification Group Services and Systems Aspects, 3GPP, TS 35.206 V 6.0.0 (2004-12), 3G Security. *Specification of the MILENAGE Algorithm Set: An example algorithm set for the 3GPP authentication and key generation functions f1, f1*, f2, f3, f4, f5, and f5*; Document 2: Algorithm Specification*, 2004.

[55] Third Generation Partnership Project, Technical Specification Universal Mobile Telecommunications System (UMTS), 3GPP, TS 33.102 V 7.0.0 (2006-05), 3G Security. *Security architecture (Release 7)*, 2006.

[56] W. Tolone, G. J. Ahn, T. Pai, and S.P. Hong. Access control in collaborative systems. *ACM Computing Surveys (CSUR)*, 37(1):29–41, March 2005.

[57] Xiaoyun Wang and Hongbu Yu. How to break md5 and other hash functions. In *Proceedings of the 24th International Conference on the Theory and Applications of Cryptographic Techniques*, pages 19–35. Springer-Verlag, May 2005.

[58] J. Yoder and J. Barcalow. Architectural patterns for enabling application security. Presented at the 4th Conference on Pattern Language Programs (PLoP97), August 1997.

A Definitions

This appendix contains a list of definitions of terms used in this report. The definitions are obtained from international standards to the extent possible, and from established sources in the literature. For terms that are defined differently in the standards, the order of prioritization is as follows: [17] first, then [24], [49], and [48].

Access control: The prevention of unauthorized use of a resource, including the prevention of use of a resource in an unauthorized manner [17].

Accessibility: The quality of being at hand and usable when needed [38].

an entity may be traced uniquely to the entity [17].

Asset: Anything that has value to the organisation [24].

Authentication: A property by which the correct identity of an entity or party is established with a required assurance. The party being authenticated could be a user, subscriber, home environment or serving network [51].

Authorization: The granting of permission based on authenticated identification [17].

Authorized: Granted rights or permissions [48].

Availability: The property of being accessible and usable on demand by an authorized entity [17, 24].

Challenge: A data item chosen at random and sent by the verifier to the claimant, which is used by the claimant, in conjunction with secret information held by the claimant, to generate a response which is sent to the verifier [19].

Claimant: An entity which is or represents a principal for the purposes of authentication. A claimant includes the functions necessary for engaging in authentication exchanges on behalf of a principal [19].

Confidentiality: The property that information is not made available or disclosed to unauthorized individuals, entities, or processes [24].

Cryptographic algorithm: An algorithm that employs the science of cryptography, including encryption algorithms, cryptographic hash algorithms, digital signature algorithms, and key agreement algorithms. [48].

Entity authentication: The corroboration that an entity is the one claimed [19].

Exclusivity: The ability to ensure access for authorized users only [38].

Identification data: Sequence of data items, including the distinguishing identifier for an entity, assigned to an entity and used to identify it [23].

Mutual authentication: Entity authentication which provides both entities with assurance of each other's identity [19].

Response: Data item sent by the claimant to the verifier, and which the verifier can process to help check the identity of the claimant [23].

Threat: A potential cause of an unwanted event, which may result in harm to a system or organisation and its assets [24].

Token: A message consisting of data fields relevant to a particular communication and which contains information that has been transformed using a cryptographic technique [19].

Trusted third party: A security authority or its agent, trusted by other entities with respect to security-related activities. A trusted third party is trusted by a claimant and/or a verifier for the purpose of authentication [19].

Unilateral authentication: Entity authentication which provides one entity with assurance of the other's identity but not vice versa [19].

Unwanted incident: Incident such as loss of confidentiality, integrity and/or availability [49].

Usable: Capable of being used [38].

Verifier: An entity which is or represents the entity requiring an authenticated identity. A verifier includes the functions necessary for engaging in authentication exchanges [19].

Vulnerability: A weakness of an asset group or group of assets, which can be exploited by one or more threats [24].

B Abbreviations

AKA Authentication and Key Agreement

AUTN Authentication Token

DAC Discretionary Access Control

ETSI European Telecommunications Standardization Institute

HTTP Hypertext Transfer Protocol

IETF Internet Engineering Task Force

IMPI IMS private identity

IMS IP multimedia subsystem

IP Internet Protocol

MAC Message Authentication Code

MD5 Message-Digest algorithm 5

OCL Object Constraint Language

PPTP Point-to-Point Tunnelling Protocol

QoS Quality of Service

RAND random challenge

RBAC Role-Based Access Control

RBM Role Based Management

RES Authentication response

SOA Service Oriented Architecture

SQN Sequence Number

UML Unified Modelling Language

VoIP Voice over Internet Protocol

C Authentication patterns

This appendix provides our full classification of authentication techniques and protocols as authentication patterns.

C.1 Classification of authentication patterns

Authentication is the process of determining who you are. More specifically, entity authentication is the corroboration that an entity is the one claimed [19]. (So in terms of the definition of a pattern given in [2] , the problem being solved is identifying an entity, and the recurring solution is authentication.) The basis of identification may be one or more of the following:

- Something the entity knows (such as a password, PIN, or secret information);
- Something the entity possesses (such as a smartcard, SIM card, or a hardware token);
- Something inherent to the entity (e.g., human physical characteristics such as fingerprints or retinal characteristics).

Authentication theory and practice has evolved over time and is well established in the literature [4, 28, 30], as well in the standards [18–20, 22, 23]. Authentication techniques are normally described as protocols. Needham and Schroeder [30] in 1978 presented some protocols for authentication in computer networks, the first major contribution to classifying techniques for authentication. Menezes, van Oorschot, and Vanstone, in their book on applied cryptography [28], provide extensive material on crypto protocols for authentication. Recently, Boyd and Mathuria [4] have published a book on authentication protocols which aims to exhaustively present each existing protocol. However, as the authors discovered, this is a formidable task, which is complicated by the fact that new protocols are still emerging to meet the needs of the changing telecommunications environment, such as the authentication protocol we have presented in [39]. Indeed, standardisation organisations assign the task of creating new protocols and algorithms for telecommunications services as needed. For example, the European Telecommunications Standardisation Institute (ETSI), has a technical group, that is responsible for creating cryptographic algorithms and protocols [42].

In order to apply authentication protocols and techniques in a model-based approach, we have classified these well-known techniques and protocols as authentication patterns specified using UML 2.0 collaborations which may be combined with services in service composition. Our classification is motivated by the need to address behavioral considerations in the patterns. This means classification based first on the service provided, unilateral authentication or mutual authentication, then based on the number of messages involved in the pattern, e.g., one message for a one pass authentication protocol, two messages for a two pass authentication protocol. The aim is to make the developer more conscious in the choice of authentication techniques to apply, while allowing flexibility with respect to the choice of protocols and algorithms and other crypto techniques to be used.

The authentication patterns we investigate are:

1. Unilateral authentication patterns:
   (a) One pass authentication

(b) Two pass authentication

2. Mutual authentication patterns:

   (a) Two pass authentication

   (b) Three pass authentication

This list may be expanded to include patterns involving trusted third parties such as adding patterns for mutual four and five pass authentication involving a trusted third party [22].

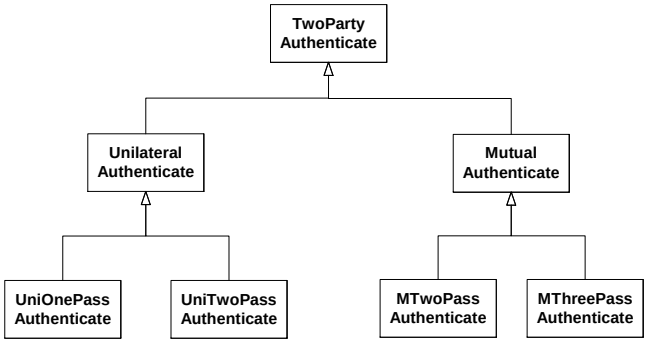

Figure 17: Authentication patterns

Fig. 17 shows our classification of authentication patterns. A generic two party authentication pattern involves communication between the two parties to establish the identity of one of the parties in the case of unilateral authentication, or both in the case of mutual authentication. Messages are generated and exchanged between the parties, at least one message/pass is required for unilateral authentication, and at least two messages/passes are required for mutual authentication. These are generic patterns that do not bind a particular protocol or algorithm. The rational behind our classification is to describe the generic pattern first based on type of authentication provided (unilateral or mutual) and number of passes/messages involved. Once a generic pattern is selected, the authentication pattern can be further differentiated in specializing the pattern depending on the type of keying, e.g., symmetric or asymmetric, to be used. The generic patterns are then further specialized with respect to the authentication technique, or cryptographic protocol and algorithm(s) to be applied, e.g., for the unilateral two-pass authentication pattern, the hypertext transfer protocol (HTTP) digest authentication protocol with the MD5 hash algorithm may be applied [14].

The unilateral one pass authentication pattern may be specialized as illustrated in the UML 2.0 class diagram shown in Fig. 18. There is a class for all unilateral one pass patterns employing symmetric crypto techniques, that is for which the authenticating party and the party requesting authentication share a common secret key, which is used in the crypto protocol. Similarly, there is a class for all unilateral one pass patterns employing asymmetric crypto techniques, and a class for all patterns for which the unilateral one pass authentication algorithm employs a hash function.

Each of these may be specialized further depending on the choice of protocol, algorithm and key size. Fig. 18 shows specializations for some standardized authentication protocols and algorithms. For example, the `UniOnePass Authenticate`

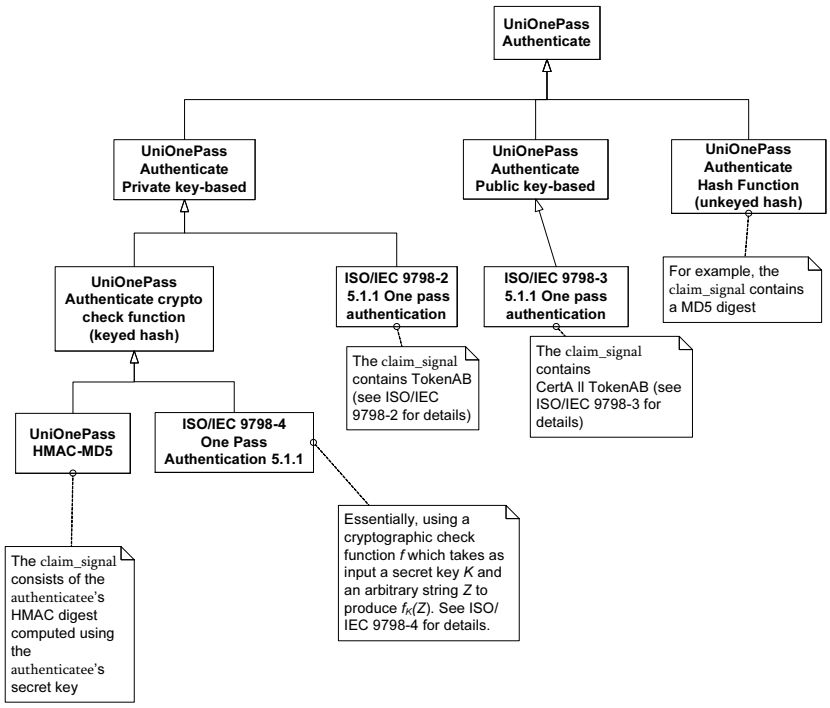

Figure 18: Classification of some unilateral one pass authentication patterns

pattern may be specialized as the `UniOnePass Authenticate` pattern using an unkeyed hash function. For this specialization, MD5, a cryptographic hash function that produces a 128-bit hash value, which is also often referred to as a message digest, may be chosen. This MD5 digest may be used in the unilateral one pass authentication pattern for authenticating a password, e.g. by using the username and password as input. However, it should be noted that authentication using a simple MD5 hash is not considered secure due to well-known attacks on the MD5 algorithm, as discussed in [57] and in [50].

Similarly, the `UniOnePass Authenticate` pattern may be specialized with respect to public key-based crypto or private key-based crypto. In the case that public key-based crypto is chosen, the ISO/IEC 9798-3 5.1.1 one pass authentication protocol may be applied [20]. In this case, as shown in Fig. 25 of Sect. C.2.1, which gives the UML2.0 interactions diagram for the unilateral one pass pattern, the `claim_signal` contains the instance playing the `authenticatee` role's public key certificate concatenated with the `TokenAB`. For details regarding the form of the token, see [20]. The token may contain a sequence number or a time stamp as a time variant parameter. An identifier for the instance playing the `authenticator` role is included in the token to ensure that the token is accepted by the intended recipient. This information is digitally signed along with some additional text using the private signature key belonging to the instance playing the `authenticatee` role. Candidate signature algorithms are chosen from e.g., the digital signature algorithm (DSA), the secure hash function (SHA) of the Rivest, Shamir, and Adleman (RSA) system or the elliptic curve DSA (ECDSA) of the elliptic curve system (ECM). For more information about these systems, and how to select the appropriate cryptographic key size, see [25].

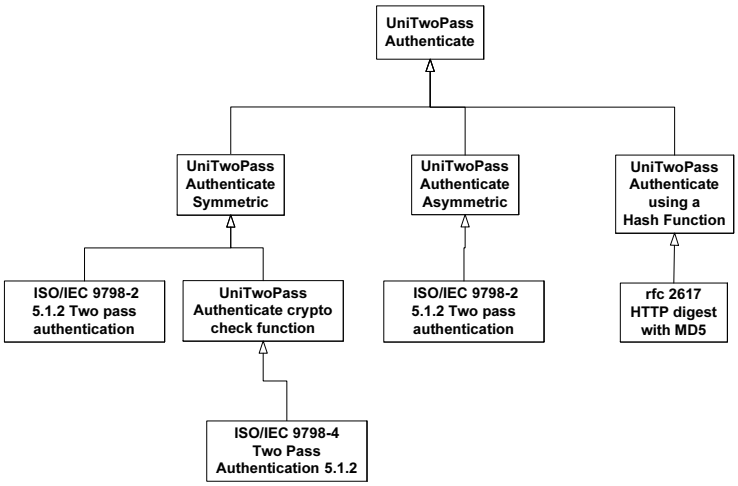

Figure 19: Classification of some unilateral two pass authentication patterns

Fig. 19 shows examples of specializations of unilateral two pass authentication patterns. The UML 2.0 interaction diagrams for the specialization of the unilateral two pass authentication using HTTP digest with MD5 as defined in [14] is shown in Fig. 31 in Section C.2.2 below. This example has been chosen as HTTP digest with MD5 is commonly used in voice over IP deployments, although the protocol and algorithm suffer from many known limitations. There are limitations simply because it is a password-based system, but also due to the known weaknesses of the MD5 algorithm [50, 57].

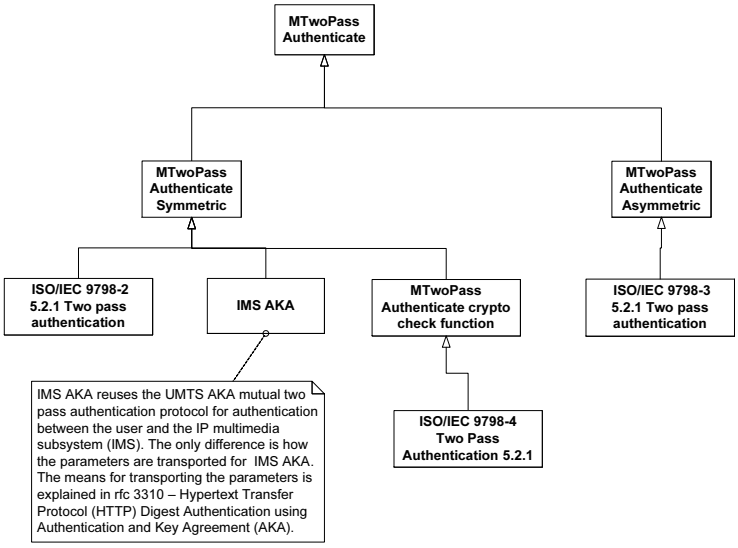

Figure 20: Classification of some mutual two pass authentication patterns

Similarly, Fig. 20 and Fig. 21 show examples of specializations of mutual two pass authentication patterns and mutual three pass authentication patterns, respectively. One

specialization of the `MTwoPassAuthenticate` pattern for symmetric key-based protocols as shown in Fig. 20 is the IP multimedia subsystem (IMS) authentication and key agreement (AKA) protocol [52]. The IMS AKA protocol was standardized by the third generation partnership project (3GPP) to provide a more secure alternative to the HTTP digest algorithm for authenticating 3GPP users to the IMS. IMS AKA extends HTTP digest by providing mutual authentication and using stronger cryptographic algorithms. IMS AKA is also stronger because it is symmetric key-based, using the symmetric key safely contained on a smart card.The UML 2.0 interaction diagrams for the specialization of the mutual two pass authentication protocol IMS AKA, is given in Fig. 37 of Section C.3.1.

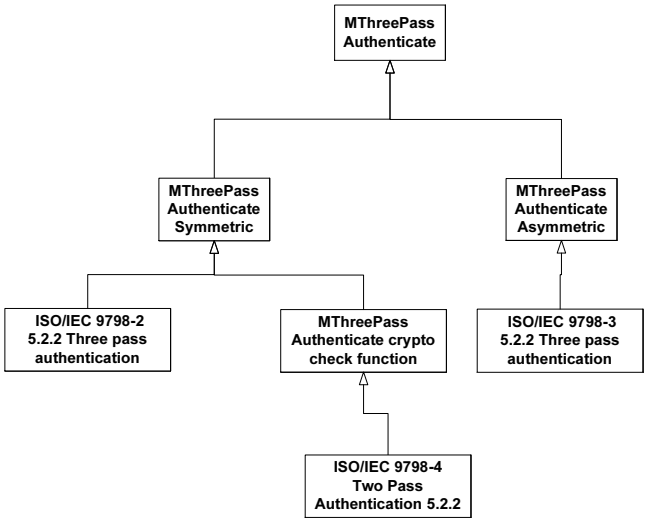

Figure 21: Classification of some mutual three pass authentication patterns

C.1.1 Choosing and applying an authentication pattern

In order to determine the generic pattern and specialisation to be applied, we need to consider what the authentication requirements for a specific service or service component are: which authentication pattern is sufficient to ensure e.g. that the user identity is verified? Is mutual authentication required? Do service requirements place restrictions on the number of passes allowed? What about the strength of the authentication required? Is single factor sufficient or is a stronger authentication required e.g. using two factor as in the case of GSM authentication? The choice of authentication pattern will depend on service requirements. Additionally, when it comes to instantiating the pattern and binding roles to agents, service requirements such as timing constraints, or maximum delay contribution due to application of the authentication pattern should be addressed. For example, a service requirement may specify that the processing involved due to the computations involved must not exceed x ms. In order to meet this requirement, the computational burden of the different algorithms must be considered. Most asymmetric crypto algorithms have very long keys, and thus cause delays which may be unacceptable. An example of a service with such strict requirements is a voice over IP service.

Choosing the appropriate authentication pattern to apply depends on several factors

including service requirements and constraints as well as different aspects of service composition design. A decision must first be made as to whether unilateral authentication is sufficient, or whether mutual authentication is required. For client server type services, it is general practice to use unilateral authentication, that is, the server authenticates the clients. However, in a multi-service provider environment, the need for authentication of the server as well should be evaluated. Or, if there is a risk of masquerading servers in the service environment, then mutual authentication should be implemented. Then, the number of passes required should be evaluated. For example, unilateral two pass authentication provides a possibility for stronger authentication than unilateral one pass authentication. This is because unilateral two pass authentication involves a remote challenge response sequence whereas unilateral one pass does not.

Once a generic pattern is selected, the authentication pattern can be further differentiated in specializing the pattern depending on the type of keying, e.g., symmetric or asymmetric, to be used. The patterns is then further specialized with respect to the authentication technique, or cryptographic protocol and algorithm(s) to be applied. A protocol may be chosen from the existing protocols available, or an original protocol may be specified. The advantage of choosing from known protocols and algorithms is that these have been subject to scrutiny and cryptanalysis, so that any weaknesses known have been published. For example, for authentication of a user to a web-based service, the unilateral two-pass authentication pattern may be chosen, using the HTTP digest authentication protocol with the MD5 hash algorithm [14].

We summarize this approach in three steps:

- First choose a generic two party authentication pattern.
- Then Choose the type of keying, e.g., symmetric or asymmetric, to be used.
- Finally, Specialize with respect to authentication technique, or crypto protocol and algorithm(s) to be applied. This step will also involve a decision on type of key, and length of key to be used.

By establishing such a stepwise selection process, we separate out the choices that must be made by the developer, and pinpoint each of the levels of specialization for awareness.

Deciding on the appropriate specialization required will involve an evaluation of the risks of abuse/misuse by unauthorized user(s), e.g. allowing an attacker to gain access to a specific service or service component. The strength of authentication required is chosen to mitigate the risks. In further work, we may provide a classification of known protocols and algorithms using this framework to assist in chosing the appropriate pattern and specialization. This classification may also classify attacks (threat scenarios) followed by the appropriate pattern(s) that may be applied to counter the attack and reduce the risk of unauthorized access.

We specify the authentication patterns for modelling in service composition using the UML 2.0 Collaboration concept [32] which provides a structured way to define services in terms of collaborating roles and a means to decompose/compose services using collaboration uses. To specify behavior associated with the collaborations we use UML 2.0 interaction [32]. The authentication patterns are modelled as UML 2.0 two party collaborations allowing reuse, and may be composed with service components or parts that are also modelled as UML 2.0 two party collaborations. In the generic patterns, we specify policies, as explained in Sect. 5, with properties/requirements on the instances playing the roles independent of choice of protocol, algorithm, keying

that are chosen in later stages of specialization. This allows for re-use of a pattern, while also allowing easy adaptation and adjustment depending on the requirements.

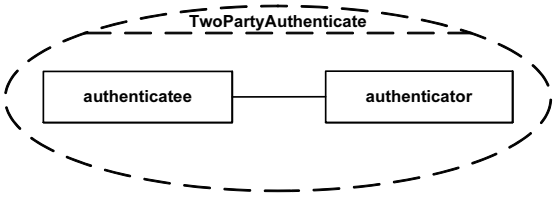

Figure 22: UM 2.0 collaboration diagram for the two party authentication pattern

A UML 2.0 collaboration diagram for the generic two party authentication pattern is given in Fig. 22. The collaboration diagram shows that the `authenticatee` role cooperates with the `authenticator` role. In the following sections, we present the specializations of the generic two party authentication pattern for the unilateral and mutual authentication patterns, modelled using UML2.0.

C.2 Unilateral authentication patterns

In unilateral authentication patterns, only one of the two parties is authenticated.

C.2.1 Unilateral one pass authentication

In this pattern, the instance playing the `authenticatee` role initiates the process and sends one message containing a `Claim`, and optionally other data (such as a public key certificate) to the instance playing the `authenticator` role. The form of the `Claim` varies depending on the crypto protocol chosen, and usually involves a time variant parameter and or a time stamp, and consists of a cleartext part, and an encrypted part. The data used as a basis for generating the `Claim` will depend on the crypto protocol to be implemented. For example, if the crypto protocol used in the implementation is the ISO/IEC 9798-2 one-pass unilateral authentication protocol, a one-pass symmetric key unilateral authentication protocol, then a shared secret key, which is used by both the instance playing the `authenticator` role and the instance playing the `authenticatee` role, and a symmetric algorithm are used [22]. The actual generation of `Claim` in an instantiation of the pattern will depend on the protocol and algorithm employed.

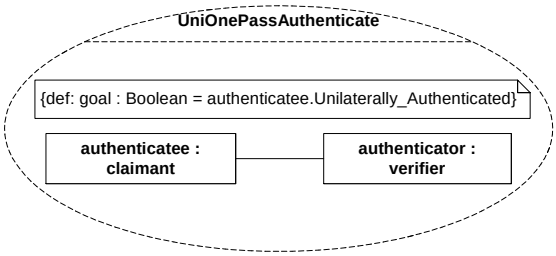

Figure 23: Unilateral one pass authentication

The collaboration diagram for the unilateral one-pass authentication pattern is given in Fig. 23. This view shows the goal for the collaboration, defined as a boolean in OCL. The goal for the pattern is that the instance playing the authenticatee role is unilaterally authenticated (by the instance playing the authenticator role). A detailed view of the pattern is give in Fig. 24. This view allows us to express more concisely the properties that the instances must have in order to participate in the pattern. Any instance playing the `authenticatee` role must possess the properties specified by `claimant` and any instance playing the `authenticator` role must possess the properties specified by `verifier`. The instance playing the `authenticatee` role must possess a secret, and the instance playing the `authenticator` role must possess `knowledge` that is mathematically related to the `secret`.

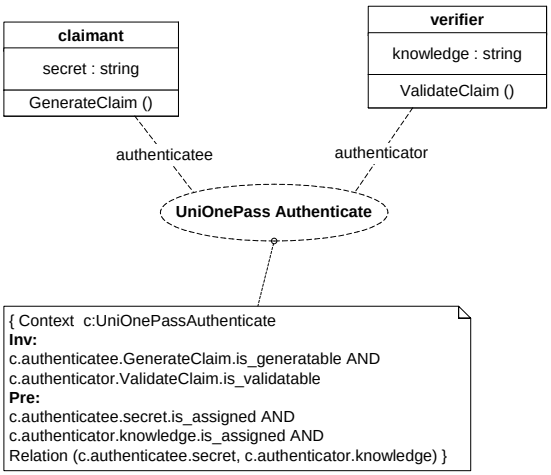

Figure 24: UML 2.0 Collaboration diagram for unilateral one-pass authentication, detailed view

Two invariants are declared: `c.authenticatee.GenerateClaim.is_generatable` and `c.authenticator.ValidateClaim.is_validatable`. The first invariant is used to check that the instance playing the the `authenticatee` role is deployed on a part of the system (terminal/node) with the required processing and computing capacity required to generate the `Claim`. Similarly, the second invariant is used to check that the instance playing the `authenticator` role is deployed on a part of the system (terminal/node) with the required processing and computing capacity required to validate the `Claim`. The reason for declaring these invariants is to ensure that the protocol and algorithm chosen are not too processor intensive for the parts on which they are deployed so that the authentication protocol can run whenever the collaboration is instantiated. The motivation for this is to ensure that service requirements regarding accessibility [38] are fulfilled when this authentication pattern is composed with service components/parts. The two pre-conditions `c.authenticatee.secret.is_assigned` and `c.authenticator.knowledge.is_assigned` check that secret and knowledge are assigned before the collaboration can instantiate. Additionally, `Relation(c.authenticatee.secret,c.authenticator.knowledge)` must evaluate to true. This means that there is a check performed to ensure that there is a pre-existing mathematical relationship between `secret` and `knowledge` as required by the authentication pattern to be deployed. The OCL pre-conditions are

used to perform a boolean check to confirm that the *a priori* conditions for the authentication protocol are fulfilled.

The purpose of these three boolean checks is to ensure that the *a priori* conditions, for the authentication protocol to be used, have been satisfied in order for the instances to successfully participate in the pattern. This means that the instances playing the `authenticatee` and `authenticator` roles have been assigned secret and knowledge correctly by the runtime system prior to service execution.

To provide information about the interactions between the two instances playing the `authenticatee` and `authenticator` roles respectively, a UML 2.0 interactions diagram is used to show the interactions in time sequence. UML 2.0 interactions uses are used for modelling authentication pattern behavior subject to service constraints such as timing, processor capacity available, or strength of algorithm required. Fig. 25 provides the interactions diagram for the generic unilateral one-pass pattern, and shows how we employ UML 2.0 interactions uses.

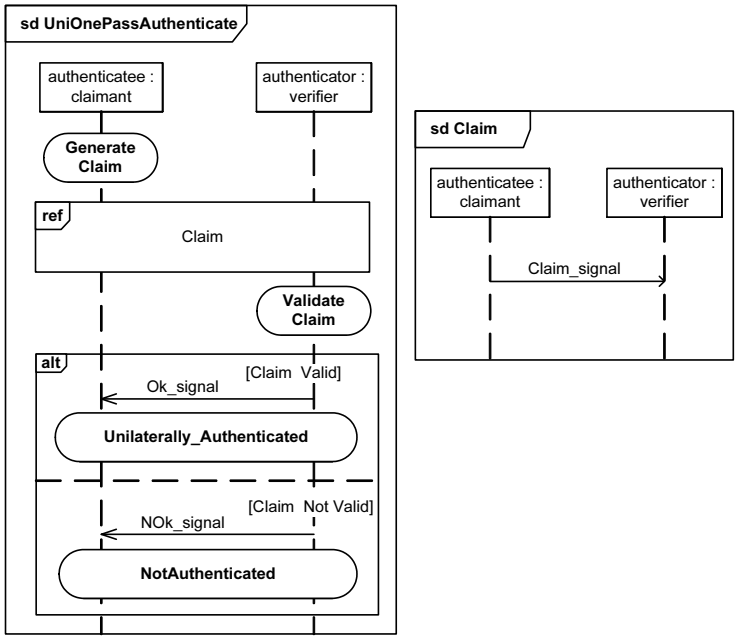

Figure 25: UML 2.0 interaction diagram for UniOnePass Authenticate

This pattern may be specialized for a particular protocol and algorithm(s) by specifying the `Claim` interactions diagram giving the detailed signal. We model the behavior involved in the so-called ”pass” using UML 2.0 interactions uses. Employing interaction uses in this manner facilitates re-usability of the patterns as well as to enable the ability to evaluate whether different crypto protocols and algorithms meet service requirements in order to obtain the combination that best suits the service requirements and other restrictions such as processor capacity in the deployed terminal when composing the pattern with services.

This flexibility allows us to test during the design phase whether a certain protocol fulfils requirements regarding strength of authentication provided. The classification of authentication into levels depending on the strength of authentication provided is out of the scope of this work, however, and we are not aware that a full classifica-

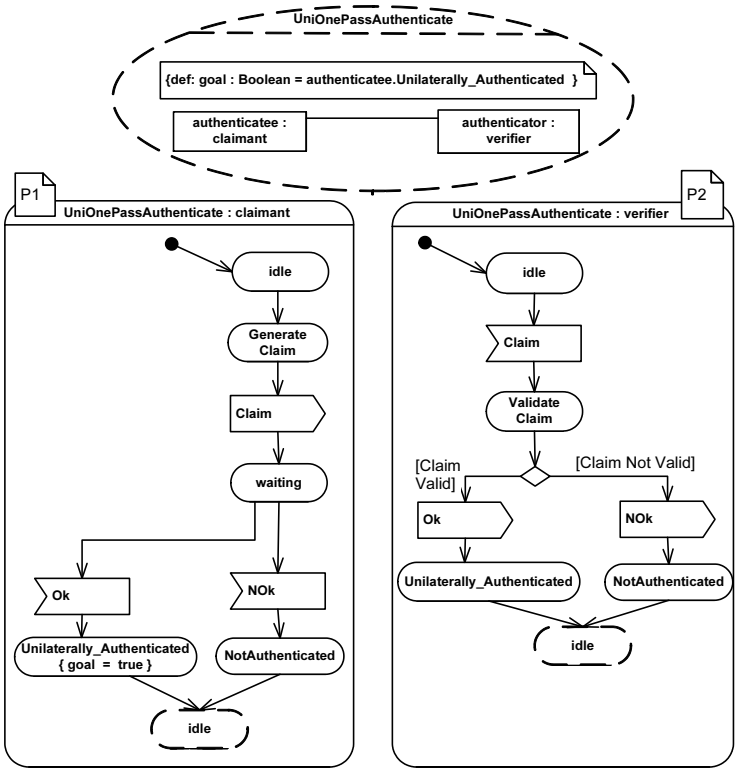

Figure 26: UML 2.0 collaboration and semantic interfaces for the `UniOnePassAuthenticate` pattern

tion of authentication protocols according to strength of authentication provided exists. However, information about strength of specific algorithms and selection of key-size for certain well-known public-key cryptography protocols and algorithms is provided in [25]. We are also interested in obtaining a suitable reference to a classification of crypto protocols, algorithms, and key-lengths used regarding processor intensivity in order to take into account timing issues in service deployments. For example, a certain protocol and algorithm run on a mobile 3G terminal is known to use x ms. The policy control check needs to determine whether x ms is within the constraints given in the role-binding policy of the service that the authentication pattern is to be composed with. If it is not within the limits, then another protocol and algorithm which is less processor intensive will need to be chosen.

Fig. 23 and Fig. 25 are used to define the semantic interfaces for the unilateral one pass authentication pattern. The semantic interfaces for the unilateral one pass authentication pattern are given in Fig. 26, defining the visible interface behavior and goals of the collaboration as explained in Sect. 4.1.2. In this case, it is a goal that the instance playing the `authenticatee` role is unilaterally authenticated by the instance playing the `authenticator` role. The two role state machines show the role behavior of the two collaboration parts participating in the pattern. `goal` $=$ `true` is an (assertion [32], not an executable property). The assertion states that the goal of unilaterally authentication has been achieved. The declarations `P1` and `P2`, in the upper corners of the two role state machines, represent the role-binding

policies for each of the two collaborating parts respectively. for the instance playing the `authenticatee` role and the instance playing the `authenticator` role. See Fig. 27 for examples of role-binding policies for each of the two collaboration parts involved in the `UniOnePassAuthenticate` pattern.

Fig. 27 provides examples of the condition parts of role-binding policies for the instances playing the `authenticatee` and `authenticator` roles respectively, and the condition part of the collaboration policy for the instantiation of the `UniOnePass-Authenticate` pattern. The role-binding policy for the `authenticatee` role states that the instance playing the `authenticatee` role must have the capacity to generate the `Claim` at any time, and must have a pre-assigned `secret`. If these two conditions are fulfilled then the instance can play the `authenticatee` role and achieve its goal. If not fulfilled, role-binding may still be allowed, but the goal will not be achieved. Similarly, the role-binding policy for the `authenticator` role states that the instance playing the `authenticator` role must have the capacity to validate the `Claim` at any time, and must have a pre-assigned `knowledge` (of the `secret`). If these two conditions are fulfilled then the instance can play the `authenticator` role. In order for the collaboration to run, both of these policies must be fulfilled.

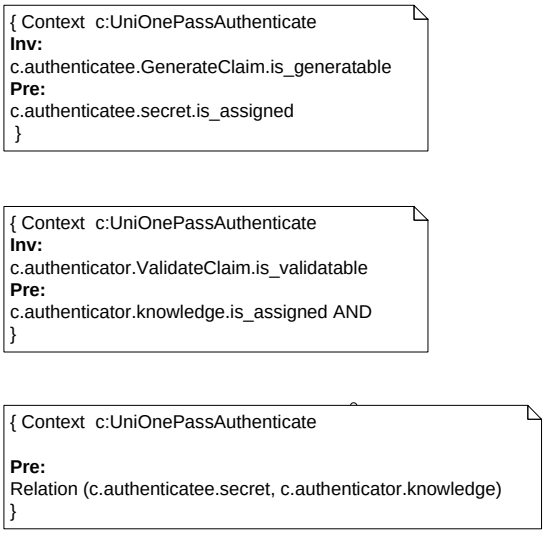

Figure 27: Examples of condition parts of role-binding policies and collaboration policy for the `UniOnePassAuthenticate` pattern

The conditions part of these policies are declared in OCL. The operation performed is a policy check: Before the role can be bound, the invariants and pre-conditions must be satisfied. Given that these conditions are satisfied, the result of the policy check is that the role can be bound.

Similarly, for the collaboration policy, before the collaboration can be instantiated, the pre-condition that there is a mathematical relationship between `secret` and `knowledge` must be evaluated. if this check results to true, and a trigger message requesting authentication is sent, then the result of the policy check is that the collaboration can be instantiated/executed.

C.2.2 Unilateral two pass authentication

In this pattern, the instance playing the `authenticator` role initiates the process and sends a challenge to the instance playing the `authenticatee` role. Upon receiving this challenge, the instance playing the `authenticatee` role generates a response and sends it back to the instance playing the `authenticator` role. The response is validated. If the response is valid, then the authentication is successful.

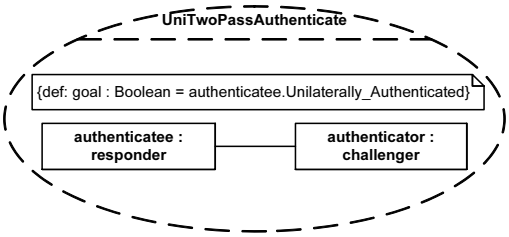

Figure 28: Unilateral two pass authentication

The instance playing the `authenticator` role sends a message containing a `Challenge`, and optionally other data (such as a public key certificate) to the instance playing the `authenticatee` role. The form of the `Challenge` varies depending on the crypto protocol chosen, and usually involves a time variant parameter and or a time stamp, and consists of a cleartext part, and an encrypted part. The data used as a basis for generating the `Challenge` will depend on the crypto protocol to be implemented. For example, if the crypto protocol used in the specialization is the ISO/IEC 9798-2 two-pass unilateral authentication protocol, a two-pass symmetric key unilateral authentication protocol, then a shared secret key, which is used by both the instance playing the `authenticator` role and the instance playing the `authenticatee` role, and a symmetric algorithm are used [22]. The actual generation of `Challenge` in an instantiation of the pattern will depend on the protocol and algorithm employed.

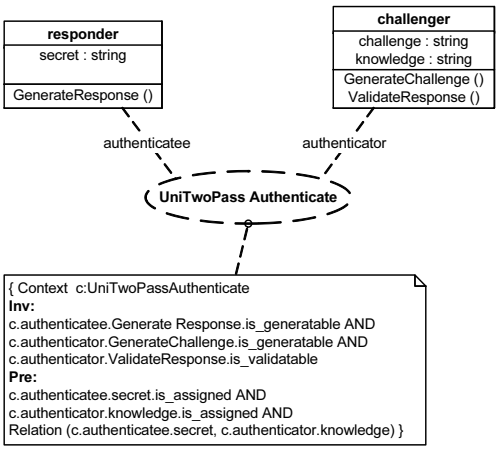

Figure 29: UML 2.0 collaboration diagram for unilateral two-pass authentication, detailed view

The collaboration diagram for the unilateral two-pass authentication pattern is given

in Fig. 28. A detailed view is given in Fig. 29. The properties of the `authenticatee` role and `authenticator` role are defined by `responder` and `challenger`. As for the unilateral one-pass pattern, the *a priori* conditions for instantiating the pattern are expressed as constraints in OCL. In this case, three invariants are declared:

```
c.authenticator.GenerateChallenge.is_generatable

c.authenticatee.GenerateResponse.is_generatable

c.authenticator.ValidateResponse.is_validatable
```

The first and third invariants are used to check that the instance playing the `authenticator` role is deployed on a part of the system (terminal/node) with the required processing and computing capacity required to generate the challenge and to validate the response. Similarly, the second invariant is used to check that the instance playing the `authenticatee` role is deployed on a part of the system (terminal/node) with the required processing and computing capacity required to generate the response. The reason for declaring these invariants is to ensure that the protocol and algorithm chosen are not too processor intensive for the parts on which they are deployed so that the authentication protocol can run whenever the collaboration is instantiated. The motivation for this is to ensure that service requirements regarding accessibility [38] are fulfilled when this authentication pattern is composed with service components/parts. The two pre-conditions `c.authenticatee.secret.is_assigned` and `c.authenticator.knowledge.is_assigned` check that `secret` and `knowledge` are assigned before the collaboration can instantiate. The third pre-condition checks that there is a mathematical relationship between `secret` and `knowledge`.

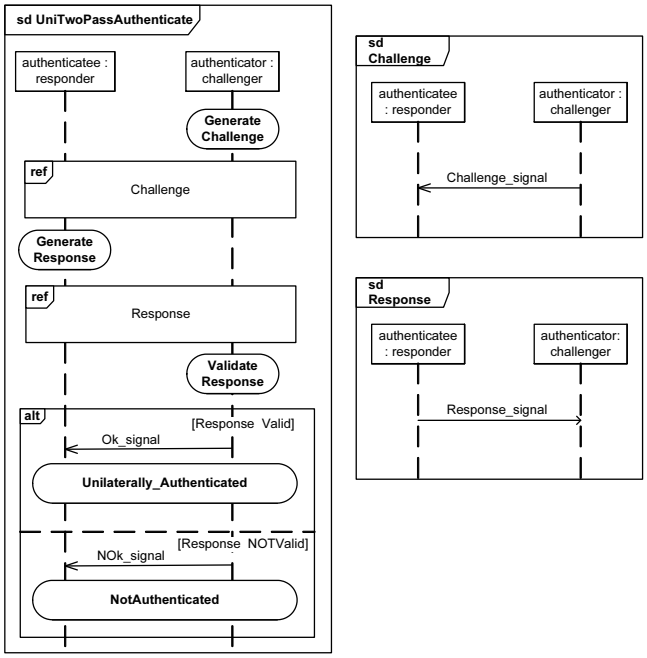

Figure 30: UML 2.0 interaction diagram for UniTwoPass Authenticate

The interaction sequences for the unilateral two pass challenge response pattern are given in Fig. 30. This diagram illustrates how we employ UML 2.0 interaction uses to enable flexibility in the specification. The challenge and response interaction uses referenced allow use of the same pattern to specify any unilateral two pass challenge response protocol. Although the intention of interaction uses is to enable reuse of a definition in many contexts, this feature also allows us to test which pair of interactions should be used for the challenge and the response passes in order to determine which best fit the requirements of the service. In this way, different protocols and algorithms may be applied subject to constraints such as timing constraints. This is to assist in selecting the protocol and algorithm that best fits the requirements. The `Generate Challenge`, `Generate Response`, and `Validate Response` state invariants depend on the protocol and algorithm chosen. In this way, the developer is able to tune the authentication pattern to fit the service requirements, and then select, freeze, and use.

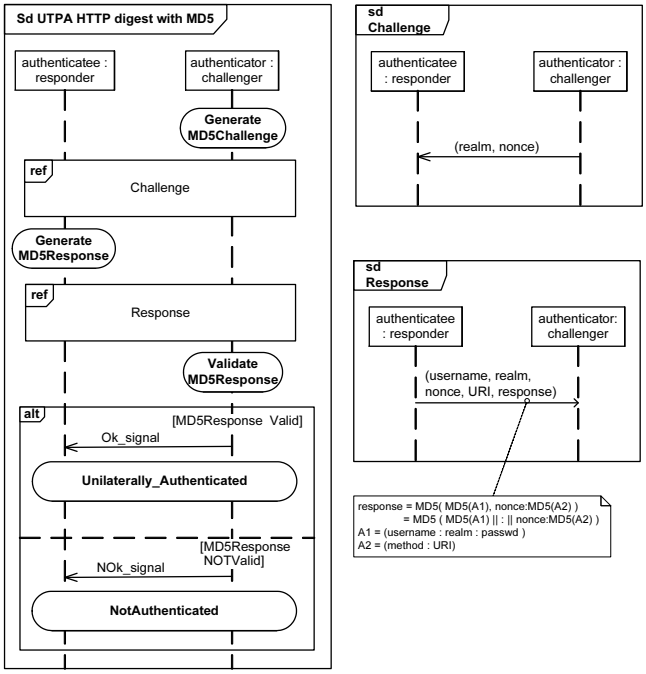

Figure 31: UML 2.0 interaction diagram for UniTwoPass Authenticate specialized for the HTTP digest protocol using the MD5 hash

A specialization of `UniTwoPass Authenticate` using the HTTP digest protocol with the MD5 algorithm is shown in Fig. 31. For this specialization, the `Challenge_signal` contains the realm and nonce values, as explained in [14]. The realm is a string displayed to the user (the instance playing the `authenticatee` role) containing at least the name of the host performing the authentication. The nonce is a data string which is uniquely generated by the instance playing the `authenticator` role. The `Response_signal` contains the username, realm, nonce, URI, and the response which is generated using the MD5 algorithm. Note that although MD5 is widely used with HTTP digest, the protocol actually specifies use of a checksum/hash function and provides an example using MD5. In other words, MD5 is not mandatory,

but is commonly used.

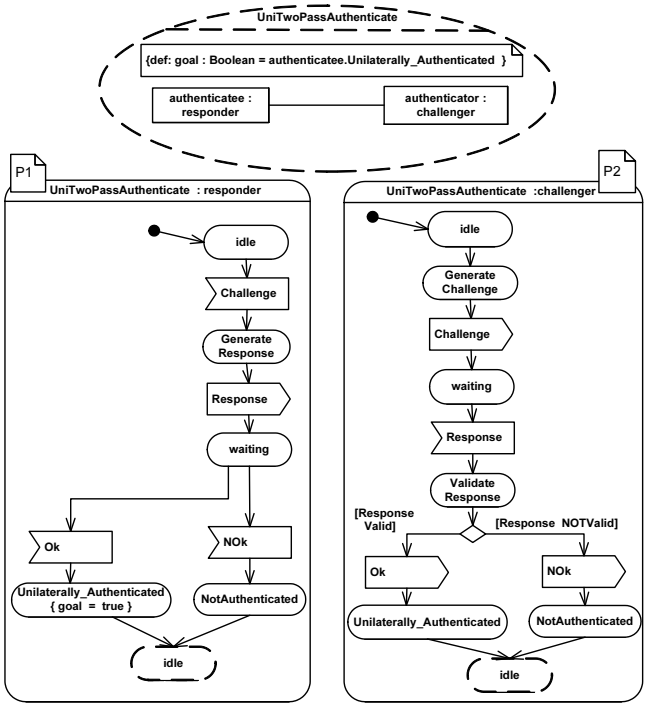

Figure 32: UML 2.0 collaboration and semantic interfaces for the `UniTwoPass Authenticate` pattern

In Fig. 32, the UML 2.0 collaboration for unilateral two-pass authentication is shown with two state machines to show the role behavior of the two collaboration parts in a unilateral two-pass challenge-response authentication pattern. These diagrams, together with goal expressions specifying properties of desirable states and events, define the semantic connector of the `UniTwoPassAuthenticate` collaboration. In addition to syntactical interfaces, semantic interfaces define the visible interface behavior and goals of the collaboration. In this case, the semantic connector defines the interface behavior and goals of the `authenticatee` and `authenticator` roles. The declarations in the upper corners of the role state machines represent the role-binding policies for each of the two collaboration parts , represented by `P1` and `P2` respectively for the instances playing the the `authenticatee` role and the instance playing the `authenticator` role.

Fig. 33 provides examples of the condition parts of role-binding policies for the instances playing the `authenticatee` and `authenticator` roles respectively, and the condition part of the collaboration policy for the instantiation of the `UniTwoPass-Authenticate` pattern. The role-binding policy for the `authenticatee` role states that the instance playing the `authenticatee` role must possess a `secret`, and it must have the capacity to be able to generate a response to the challenge sent by the `authenticator`. If these conditions are fulfilled, then the instance can play the `authenticatee` role. Similarly, the instance playing the `authenticator` role must possess `knowledge` (that is mathematically related to the `secret` assigned to the instance playing the `authenticatee` role), and it must be able to to generate a

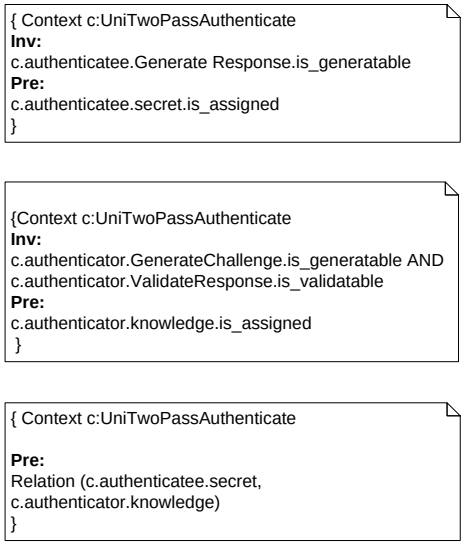

Figure 33: Examples of the condition parts of role-binding policies and collaboration policy for the `UniTwoPassAuthenticate` pattern

`challenge`, which is sent to the `authenticatee`, and to validate the reponse. In order for the collaboration to run, both of the role-binding policies must be fulfilled. The condition part of the collaboration policy states that in order for the collaboration to instantiate, there must be a mathematical relationship between `secret` and `knowledge`.

C.3 Mutual authentication patterns

In mutual authentication patterns, both of the two parties are authenticated.

C.3.1 Mutual two pass authentication

In this pattern, the instance playing the `authenticateeA` role initiates the process and sends a message containing a claim, `ClaimAB`, and optionally other data (such as a public key certificate) to the instance playing the `authenticateeB` role. The the instance playing the `authenticateeB` role must first validate this `ClaimAB`, and if it is valid, then a claim, `ClaimBA`, is generated by the instance playing the `authenticateeB` role and sent to the instance playing the `authenticateeA` role. The instance playing the `authenticateeA` role must then validate `ClaimBA`, and if it is also valid, then the instances are mutually authenticated.

The form of the claims generated varies depending on the crypto protocol chosen, and usually involves a time variant parameter and or a time stamp, and consists of a cleartext part, and an encrypted part. The data used as a basis for generating the `ClaimAB` and the `ClaimBA` will depend on the crypto protocol to be implemented. For example, if the crypto protocol used in the implementation is the ISO/IEC 9798-2 two-pass mutual authentication protocol, a two-pass symmetric key mutual authentication protocol, then a shared secret key, which is used by both the instance playing the `authenticator` role and the instance playing the `authenticatee` role, and a

symmetric algorithm are used [22]. The actual generation of `ClaimAB`, `ClaimBA` in an instantiation of the pattern will depend on the protocol and algorithm employed.

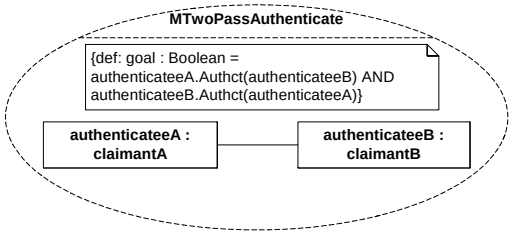

Figure 34: Mutual two pass authentication

The collaboration diagram for the mutual two-pass authentication pattern is given in Fig. 34. The properties of the `authenticateeA` role and `authenticateeB` role are defined by `claimantA` and `claimantB`. This view shows the goal for the collaboration, defined as a boolean in OCL. The goal for the pattern is that the instance playing the `authenticateeA` role is authenticated (by the instance playing the `authenticateeB` role) and that the instance playing the `authenticateeB` role is authenticated (by the instance playing the `authenticateeA` role). A detailed view is given in Fig. 35. As for the unilateral authentication patterns described above, the *a priori* conditions for instantiating the pattern are expressed as constraints in OCL. This is to establish that all of the requirements for the authentication protocol to run successfully are fulfilled prior to running the authentication protocol. For example, if the instance playing the `authenticateeA` role has not been assigned a `secretA` ahead of time, then the authentication process will error when run. All of these conditions must be established ahead of time for the authentication protocol to be able to run. Having established *a priori* conditions, authentication can be performed whenever authentication is required e.g. in a service collaboration.

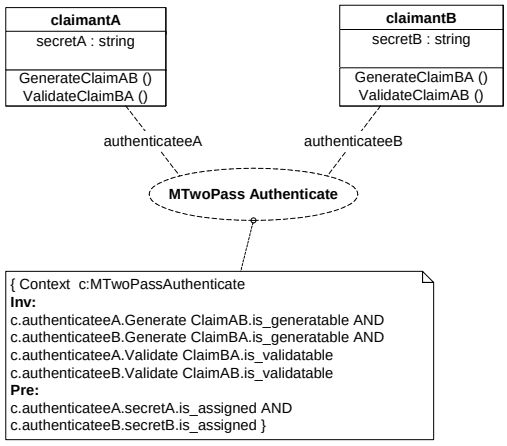

Figure 35: UML 2.0 collaboration diagram for mutual two-pass authentication, detailed view

In this case, five invariants are declared as OCL constraints. The first, third and fifth invariants are used to check that the instance playing the the `authenticateeB`

role is deployed on a part of the system (terminal/node) with the required processing and computing capacity required to generate the challenge, `ChallengeAB`, to generate the claim, `ClaimBA`, and to validate the response, `ClaimAB`. Similarly, the second and fourth invariants are used to check that the instance playing the `authenticateeA` role is deployed on a part of the system (terminal/node) with the required processing and computing capacity required to generate the response `ClaimAB` and to validate the response `ClaimBA`. The reason for declaring these invariants is to ensure that the protocol and algorithm chosen are not too processor intensive for the parts on which they are deployed so that the authentication protocol can run whenever the collaboration is instantiated. The motivation for this is to ensure that service requirements regarding accessibility [38] are fulfilled when this authentication pattern is composed with service components/parts. The two pre-conditions `c.authenticateeA.secretA.is_assigned` and `c.authenticateeB.secretB.is_assigned` check that `secretA` and `secretB` are assigned before the collaboration can execute. It is also possible to state pre-conditions checking that the instance playing `authenticateeA` has knowledge of `secretB` and that the instance playing `authenticateeB` has knowledge of `secretA` to ensure that all of the *a priori* conditions are satisfied.

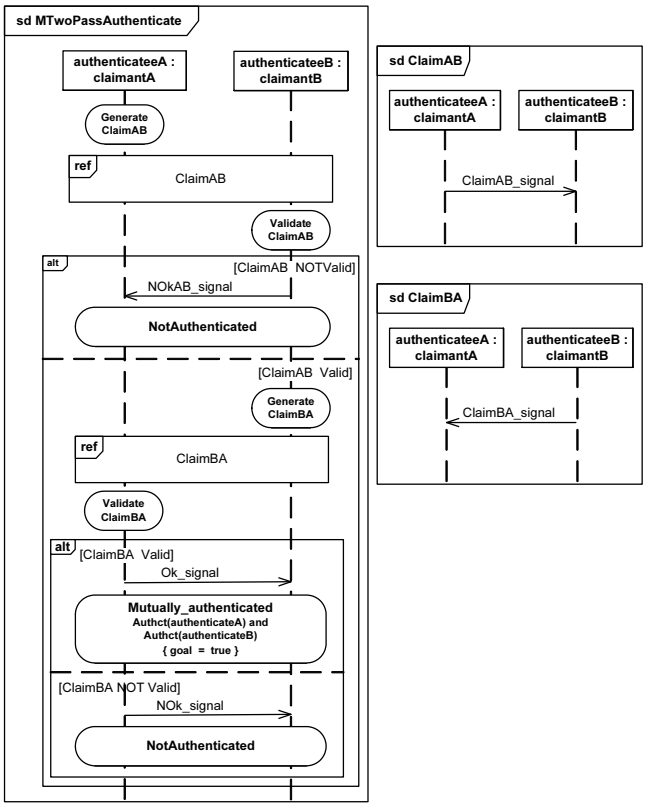

Figure 36: UML 2.0 interaction diagram for MTwoPass Authenticate

The interaction sequences for the `MTwoPass Authenticate` pattern are given in Fig. 36. The challenge and response interaction uses referenced allow re-use of the same pattern to specify any mutual two pass challenge response protocol.

In the following, we demonstrate specialization of the UML 2.0 interactions dia-

gram for the mutual two pass authentication pattern, shown in Fig. 36, for the IMS AKA protocol. The IMS AKA protocol reuses the UMTS AKA mutual two pass authentication protocol [55] for authentication between the user and the IP multimedia subsystem (IMS). The only difference is in how the parameters are transported for IMS AKA. The means for transporting the parameters is explained in [31]. Essentially, IMS AKA extends the security of the hypertext transfer protocol (HTTP) Digest. Although HTTP digest is a unilateral two-pass authentication protocol (username and password-based) with known weaknesses, the IMS AKA extension improves the protocol by providing mutual authentication and the use of stronger algorithms. Furthermore, the symmetric key used in the protocol is safely contained on the UMTS integrated circuit card (UICC) [52].

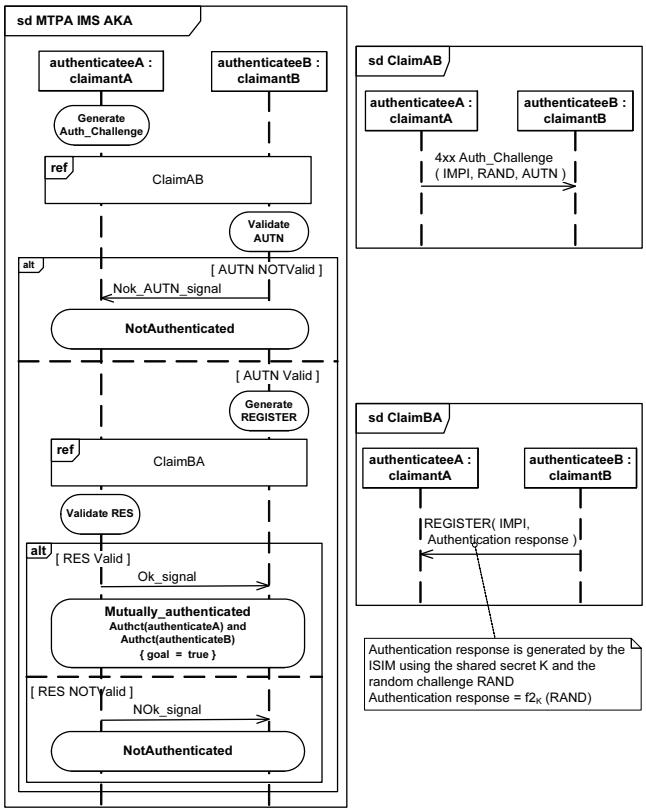

Figure 37: UML 2.0 interaction diagram for MTwoPass Authenticate specialized for IMS AKA, detailed view

In this specialization of the pattern, as shown in Fig. 37, the instance playing the `authenticateeA` role initiates the process and sends a challenge message containing the IMS private identity (IMPI), a random challenge (RAND), and the authentication token (AUTN) to the instance playing the `authenticateeB` role. The AUTN, which contains a message authentication code (MAC) and the sequence number (SQN), is used to authenticate the instance playing the `authenticateeA` role. Then the instance playing the `authenticateeB` role must first validate the AUTN, and if it is valid, then an authentication response (RES) is generated by the instance playing the `authenticateeB` role and sent to the instance playing the `authenticateeA` role

along with the IMPI. The RES is generated using the shared secret key K and RAND and employs the UMTS security algorithm $f2$, a message authentication function as specified in [53] and in [54]. The instance playing the `authenticateeA` role must then validate this RES, if it is also valid, then the instances are mutually authenticated.

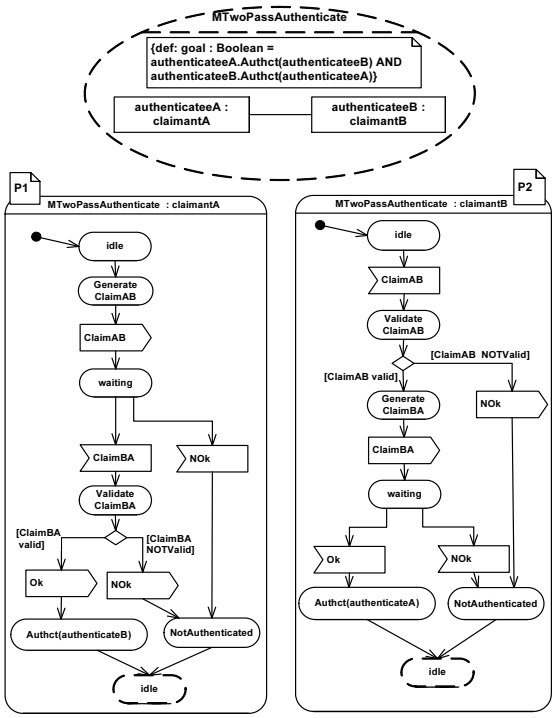

Figure 38: UML 2.0 collaboration and semantic interfaces for the `MTwoPass Authenticate` pattern

The semantic connector for the mutual two pass authentication pattern is given in Fig. 38, defining the visible interface behavior and goals of the collaboration as explained in Sect. 4.1.2.

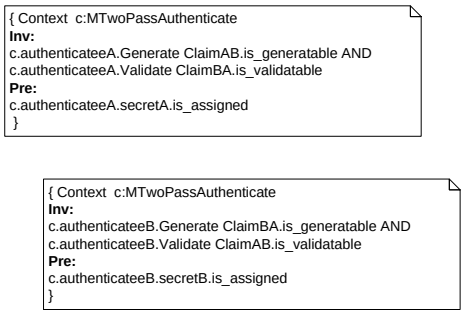

Figure 39: Examples of the condition parts of role-binding policies for the `MTwoPassAuthenticate` pattern

Fig. 39 provides examples of the condition parts of role-binding policies for the

generic MTwoPassAuthenticate pattern. The condition part of the role-binding policy shown for the instance playing the authenticateeA role states that the instance playing the authenticateeA role must possess a secretA, and it must have the capacity to be able to generate ClaimAB which will be sent to the instance playing the authenticateeB role, and it must have the capacity to validate the ClaimBA received from the instance playing the authenticateeB role. If these conditions are fulfilled, then the instance can play the authenticateeA role. Similarly, the condition part of the role-binding policy shown for the instance playing the authenticateeB role states that the instance must possess a secretB, and it must be able to to generate a ClaimBA, which is sent to the authenticateeA, and it must be able to validate the ClaimAB received from the instance playing the authenticateeA role. If these conditions are fulfilled, then the instance can play the authenticateeB role. In order for the collaboration to run, both of the role-binding policies must be fulfilled.

C.3.2 Mutual three pass authentication

In this pattern, the instance playing the authenticateeB role initiates the process and sends a sends a challenge to the instance playing the authenticateeA role. Upon receiving this challenge, the instance playing the authenticateeA role generates a response and sends it back to the instance playing the authenticateeB role. The response is validated. If the response is valid, then the instance playing the authenticateeB role generates a claim which is sent to the instance playing the authenticateeA role. The instance playing the authenticateeA role must then validate this claim, if it is also valid, then the instances are mutually authenticated.

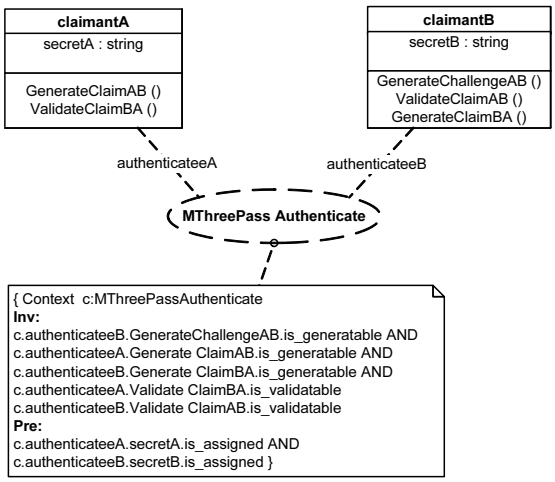

Figure 40: UML 2.0 collaboration diagram for mutual three-pass authentication, detailed view

A detailed view of the collaboration diagram for the mutual three-pass authentication pattern is given in Fig. 40. The properties of the authenticateeA role and authenticateeB role are defined by claimantA and claimantB. This view shows the goal for the collaboration, defined as a boolean in OCL. The goal for the pattern is that the instance playing the authenticateeA role is authen-

ticated (by the instance playing the `authenticateeB` role) and that the instance playing the `authenticateeB` role is authenticated (by the instance playing the `authenticateeA` role).

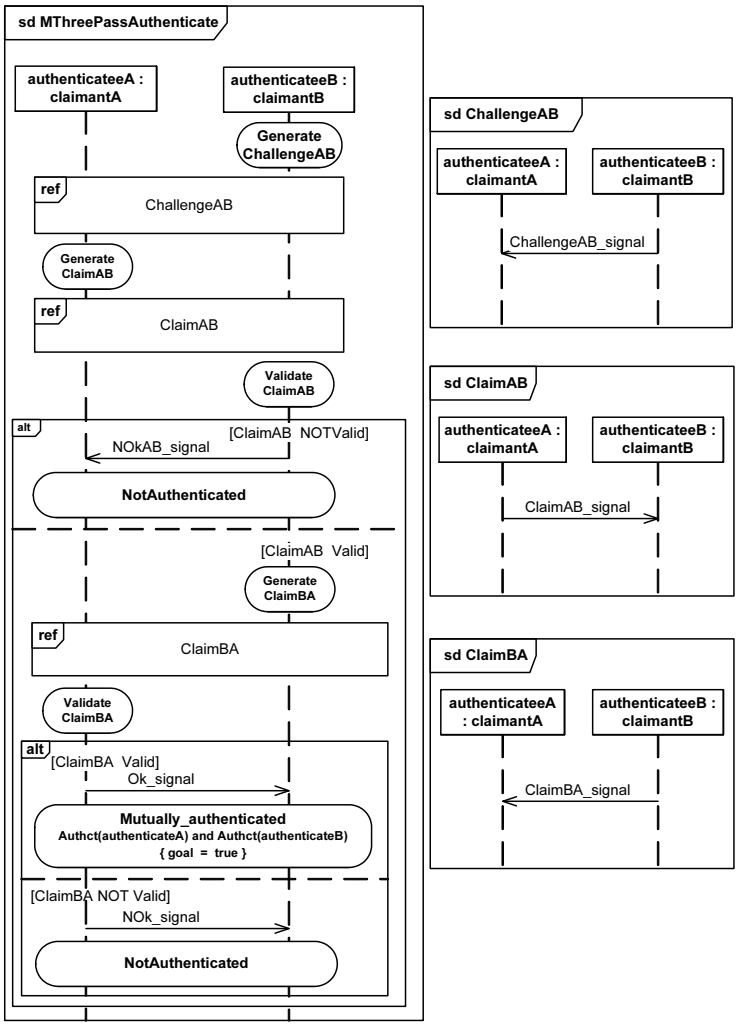

Figure 41: UML 2.0 interaction diagram for MThreePass Authenticate

The interaction sequences for the mutual three-pass challenge response pattern are given in Fig. 41. As for the other authentication patterns we have discussed above, the interactions diagram illustrates how we employ UML 2.0 interaction uses to enable reuse and flexibility. The `ChallengeAB`, `ClaimAB`, and `ClaimBA` interactions uses referenced allow use of the same pattern to specify any mutual three-pass authentication pattern.

The semantic connector for the mutual three-pass authentication pattern is given in Fig. 42, defining the visible interface behavior and goals of the collaboration as explained in Sect. 4.1.2. The goal for the collaboration is that the instance playing the `authenticateeA` role is authenticated by the instance playing the `authenticateeB` role, and that the instance playing the `authenticateeB` role is authentic-

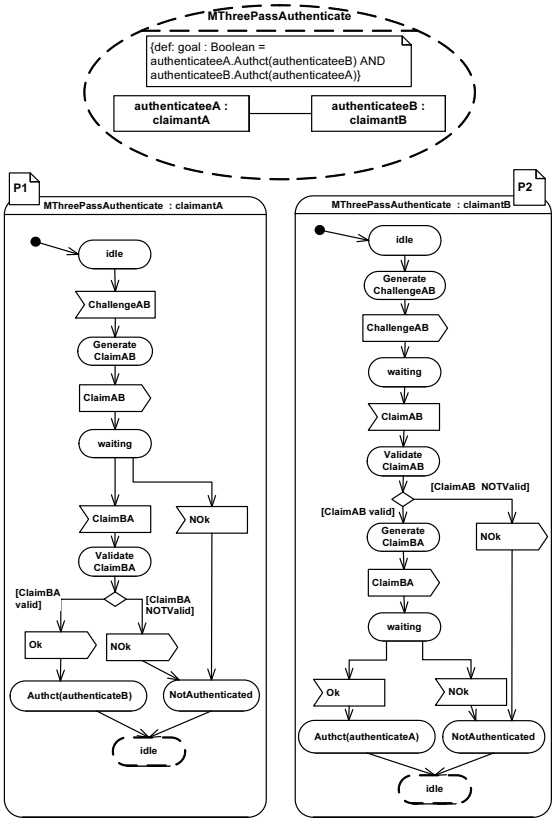

Figure 42: UML 2.0 collaboration and semantic interfaces for `MThreePass Authenticate`

ated by the instance playing the `authenticateeA` role.

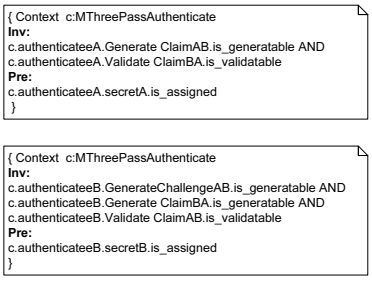

Figure 43: Role-binding policies for the `MThreePassAuthenticate` pattern

Fig. 43 provides examples of the condition parts of role-binding policies for the generic `MThreePassAuthenticate` pattern. The condition part of the role-binding policy for the instance playing the `authenticateeA` role states that the instance must possess a `secretA`, and it must have the capacity to generate `ClaimAB`, which will be sent to the instance playing the `authenticateeB` role, and it must be able to validate `ClaimBA`. If these conditions are satisfied, then the instance can play the

authenticateeA role. Similarly, the condition part of the role-binding policy for the instance playing the authenticateeB role states that the instance must possess a secretB, and it must have the capacity to generate a ChallengeAB, which will be sent to the instance playing the authenticateeA role, and be able to validate the ClaimAB received from the instance playing the authenticateeA role. The instance must also be able to generate the ClaimBA, which is sent to the instance playing the authenticateeA role. If these conditions are fulfilled, then the instance can play the authenticateeB role.

D Authorization patterns

This appendix presents the full classification of authorization patterns.

In order to describe any authorization pattern, it is important to recognize that any authorization pattern requires that authentication has been performed before any authorizations may be granted. Authentication and authorization patterns are combined to describe how access rights are granted and are thus essential to access control. Additionally, an access control model is required for access rights administration. Well known examples of access control models are e.g., discretionary access control, mandatory access control, role-based access control, and others [12].

In general, systems are deployed with a wide range of applications and services each having different authentication and authorization requirements. Authentication and authorization can be designed and deployed for each service separately, on an individual basis. Each authentication and authorization solution can be deployed and maintained separately and independently for each service, however, this leads to the development of parallel solutions, which would not be cost efficient if a single service provider offers a range of different services. In the case that several services are being offered, it is desirable to manage authorizations and to some extent authentication also, in a centralized manner.

In service composition, we require the possibility to specify authentication requirements and authorization requirements depending on the individual services. Centralized management of authorizations is important in the environment of dynamic composition of services in order to manage access rights efficiently to enable authorization for use in a wide range of service collaborations. We therefore consider authorization patterns that allow for handling of authorizations in a centralized manner.

Although there are many authorization management solutions for managing authorizations these have essentially been classified as two basic authentication and authorization architectures [12]:

- *User Pull*: Authentication is performed by an access server, which also issues authorizations to the user. The user then presents authorizations directly to the service.

- *Server Pull*: The service centralizes information about user entity authorizations on an access server. The service authenticates the user. When the user attempts to access the service, the service queries the access server to determine whether the user is authorized.

The user-pull and server-pull authorization architectures were first identified for application to web-based solutions in [34], and extended in [12] for application to any application or service that a user interacts with such as email-servers, Web sites, services or any system that requires authentication and authorization.

These architectures provide a means for handling authorizations in a centralized manner. The role of access server is played for e.g. issuing and storing authorizations associated with the user role. How the authorizations are activated and administered is described by the access control model to be deployed such as role-based access control (RBAC).

D.1 Userpull

The UML 2.0 collaboration structure diagram for the User Pull authentication and authorization architecture is given in Fig. 44. The collaboration diagram shows that the `Access Server` role collaborates with the `User` role, and the `User` role collaborates with the `Service` role. Authentication is performed by the instance playing the `Access Server` role, which also issues authorizations and optionally authentication information to the instance playing the `User` role. The instance playing the `User` role presents authorizations directly to the instance playing the `Service` role. In some specializations of this pattern the service may additionally require that the user authenticates to the service.

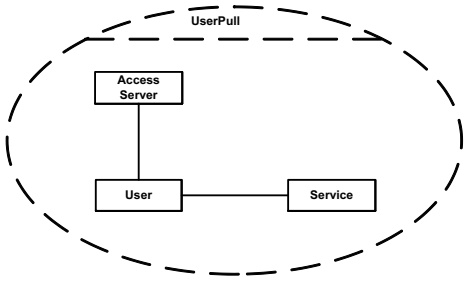

Figure 44: UserPull patterns

In order to facilitate composition of AA-patterns and services, we model the behavior required by the `Service` role in the `UserPull` collaboration separately from the ”pure” service behavior. By doing this, we avoid modifying the ”pure” service role. We therefore rename the `Service` role in `UserPull` authentication and authorization patterns, naming it the `Service Access Filter`. This participant, the `Service Access Filter`, between the `Service` role and the `User`, performs the checking of authorizations to determine if the instance playing the `User` role is allowed to access the service. In this way, incrementality can be achieved by allowing services to be defined and developed separately, at different times independently and then composed with AA-patterns.

We model the User Pull authentication and authorization patterns as a UML 2.0 collaboration that defines three collaborating participants that interact to implement the user pull authentication and authorization behavior: these are the `User`, `Access Server`, and `Service Access Filter` roles. Application of certain AA-patterns to the User Pull services is represented by three collaboration uses as illustrated in Fig. 45 and explained in the following:

- `TwoPartyAuthenticate:` This pattern, which we have modelled as a UML 2.0 collaboration in Fig. 22, is shown in Fig. 45 bound to the `User` and `Access Server` roles. Here, the `authenticatee` role is bound to the `User` role, and the `authorisor` role is bound to the `Access Server` role. For the instantiation of this pattern, it is expected that an appropriate two party authentication pattern is chosen and applied from the set of authentication patterns described in Sect. C.

- `Auths Activation:` This pattern consists of a request by the instance playing the `authsrequestor` role for authorizations to be activated and sent to the

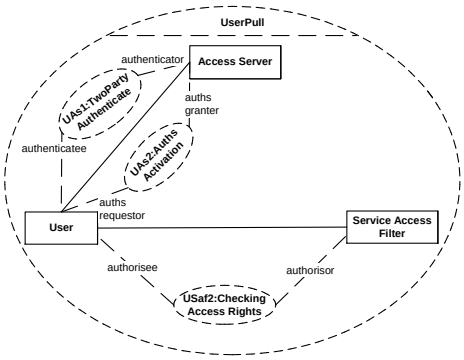

Figure 45: UserPull patterns modified

instance playing the `authsrequestor` role. The authorizations govern which services the user is allowed to access. The way in which the authorizations are activated depends on the access control model that is used. This pattern is invoked after the collaboration `TwoPartyAuthenticate` has reached its goal of e.g., unilaterally authenticating the `authenticatee`. In the `UserPull` collaboration, `Auths Activation` is shown bound to the `User` and `Access-Server` roles.

- `Checking Access Rights:` This pattern is invoked whenever the instance playing the `User` role requests access to a service. The instance playing the `authoriser` role then checks the authorizations to establish whether the instance playing the `User` role shall be granted access to the service. In the `User Pull` collaboration, `Checking Access Rights` is shown bound to the `User` and `Service Access Filter` roles.

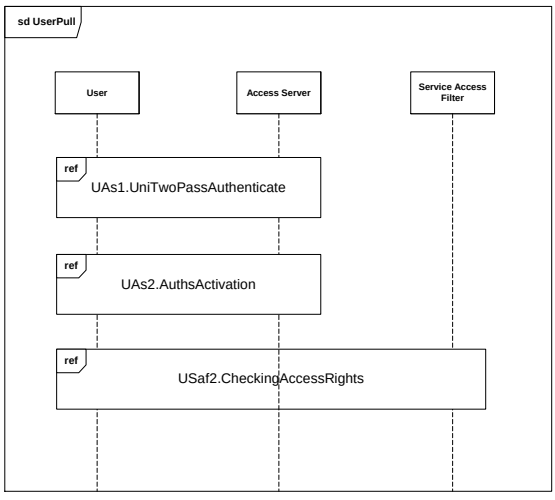

Figure 46: Composition of AA-patterns in UserPull, interactions overview

A UML2.0 interactions overview showing the composition of the instances of the

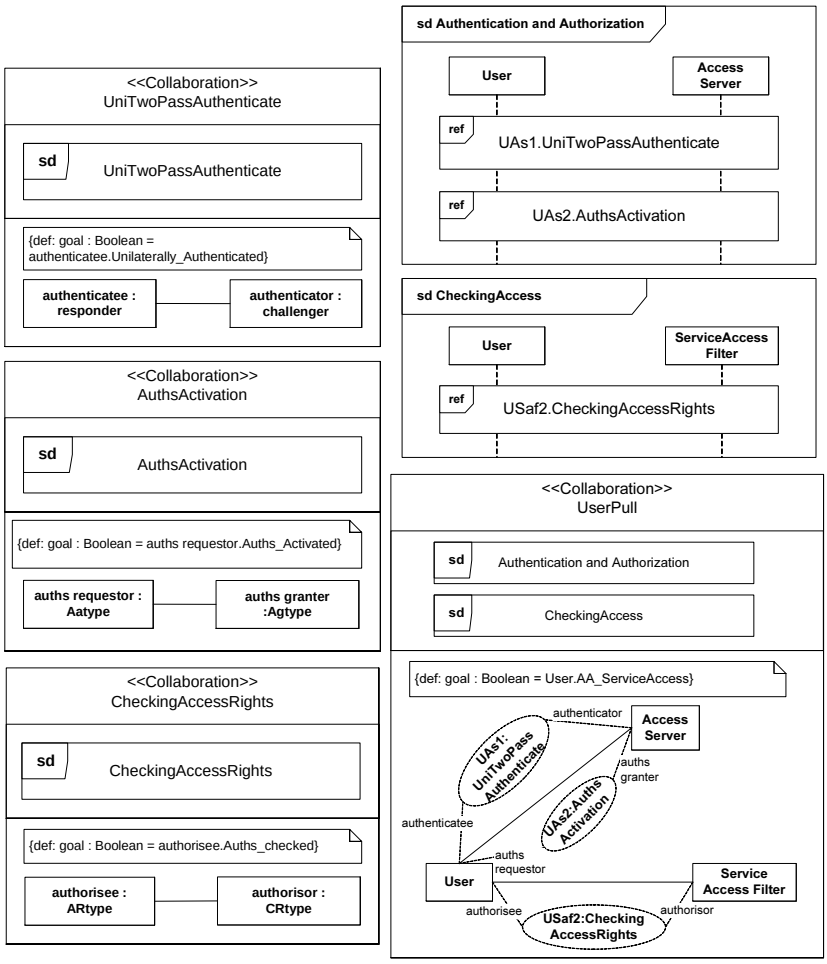

Figure 47: Composition of AA-patterns in UserPull

interactions that describe the UserPull patterns behavior is given in Fig. 46. This interactions overview shows the sequencing of the interactions, however, it doesn't provide the structural information that is shown in Fig. 47.

Fig. 47 shows the three collaborations involved in the User Pull pattern along with references to the UML2.0 interactions for modelling the behavior associated with each of the collaborations. In this figure, collaboration uses are employed to make the interactions of the collaborations available in the `UserPull` collaboration. The collaboration use of `UAs1` of `UniTwoPass Authenticate` binds the `authenticatee` and `authenticator` roles to `User` and `Access Server` respectively. Similarly, the collaboration use `UAs2` of `Auths Activation` binds the `auths requestor` and`auths granter` roles to `User` and `Access Server` respectively. The collaboration use `USaf2` of `Checking access rights` binds the `authorisee` and `authorisor` roles to the `User` and `Service Access Filter` roles respectively. Goals expressions are defined for each of the collaborations separately, as well as for the composition of these in `UserPull`. Dynamic linking of the interactions of the structural parts can be expressed using composition policies as explained in Section 6.

D.2 Serverpull

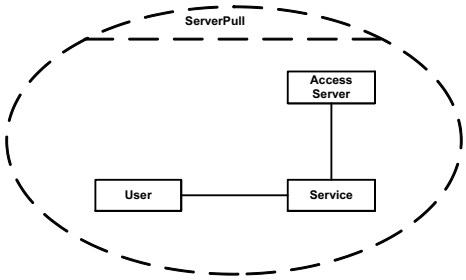

Figure 48: ServerPull patterns

The UML 2.0 collaboration structure diagram for the Server Pull authentication and authorization architecture is given in Fig. 48.

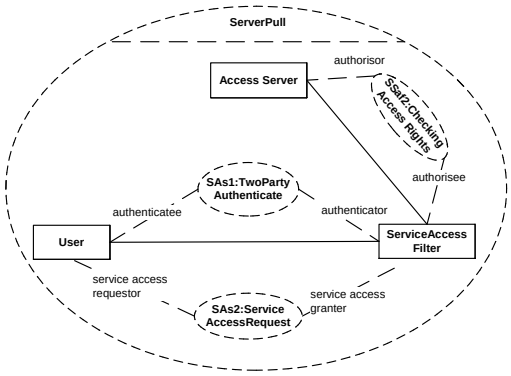

Figure 49: ServerPull patterns

Fig. 49 shows the `ServerPull` authentication and authorization services modelled as a UML 2.0 collaboration that defines three collaborating participants that interact to implement the `ServerPull` authentication and authorization behavior: these are the `User`, `Access Server`, and `Service Access Filter` roles. Application of certain AA-patterns to the `ServerPull` services is represented by three collaboration uses as illustrated in Fig. 49 and explained in the following:

- `TwoPartyAuthenticate:` This pattern, which we have modelled as a UML 2.0 collaboration in Fig. 22, is shown in Fig. 49 bound to the `User` and `Service Access Filter` roles. Here, the `authenticatee` role is bound to the `User` role, and the `authorisor` role is bound to the `Service Access Filter` role. For the instantiation of this pattern, it is expected that an appropriate two party authentication pattern is chosen and applied as described in Sect. 3.1. and further explained in Sect. 4.
- `Service Access Request:` This pattern consists of a request by the instance playing the `service access requestor` role for access to the service. This pattern is invoked after the collaboration `TwoPartyAuthenti-`

cate has reached its goal of e.g., unilaterally authenticating the authenticatee. In the Server Pull collaboration, Service Access Request is shown bound to the User and Service Access Filter roles.

- Checking Access Rights: This pattern is invoked whenever the instance playing the User role requests access to a service. The Access Server, which is the instance playing the authoriser role checks the authorizations to establish whether the instance playing the User role shall be granted access to the service. In the Server Pull collaboration, Checking Access Rights is shown bound to the Service Access Filter roles and Access Server roles.

In order to facilitate composition of the authentication and authorization patterns used in Server Pull with services, we require that the behavior defined by the Service role in the Server Pull collaboration is modelled separately from the "pure" service behavior. The separation is desirable in order to achieve incremental service development. We therefore rename the Service role of Fig. 48 to Service Access-Filter role so that we can distinguish this role behavior from the "pure" service roles involved in a collaboration of AA-patterns and service roles.

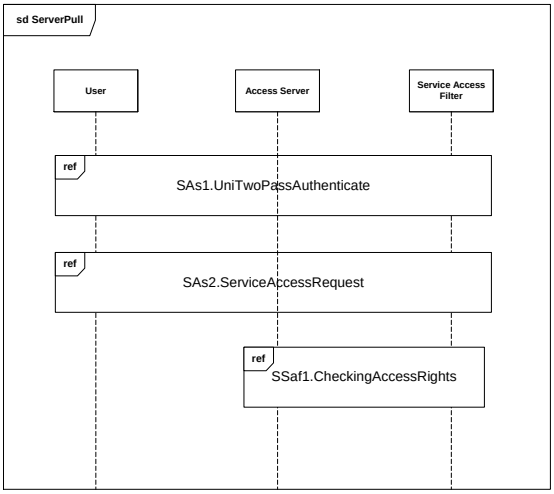

Figure 50: Composition of AA-patterns in ServerPull, interactions

A UML2.0 interactions overview showing the composition of the instances of the interactions that describe the UserPull patterns behavior is given in Fig. 50. The interactions overview shows the sequencing of the interactions, however, it doesn't provide the structural information that is shown in Fig. 51.

Fig. 51 shows the three collaborations involved in the Server Pull pattern along with references to the UML2.0 interactions for modelling the behavior associated with each of the collaborations. In this figure, collaboration uses are employed to make the interactions of the collaborations available in the UserPull collaboration. The collaboration use of SAs1 of UniTwoPass Authenticate binds the authenticatee and authenticator roles to User and Service Access Filter respectively. Similarly, the collaboration use SAs2 of Service Access Request binds the service accesss requestor andservice access granter roles to User and

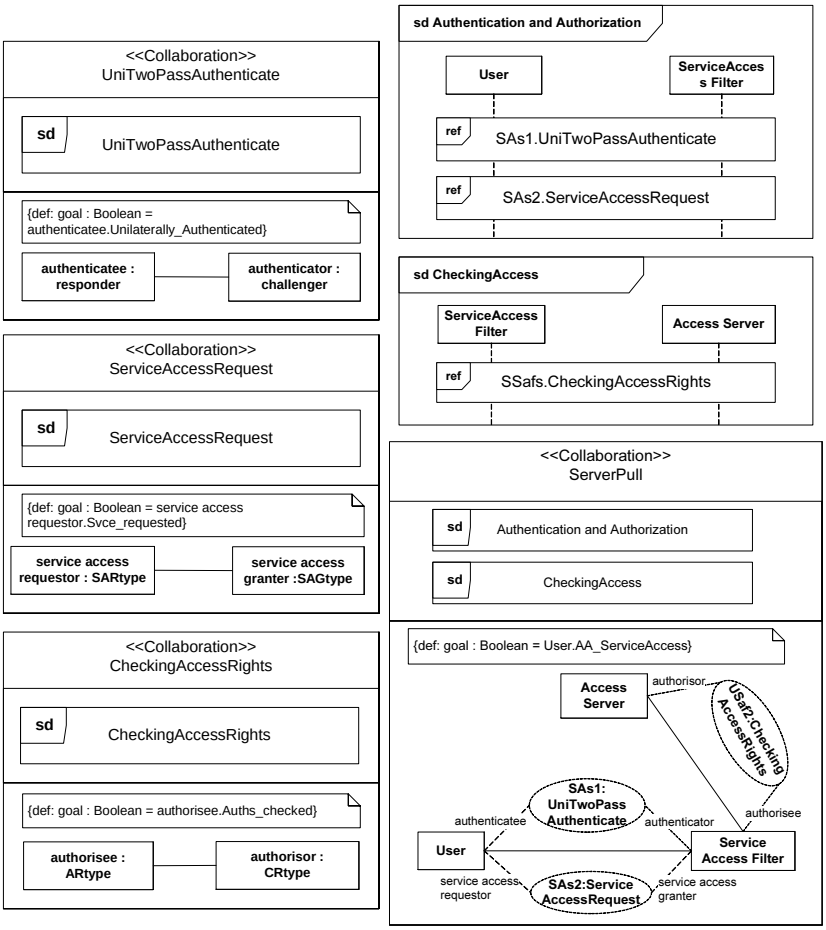

Figure 51: Composition of AA-patterns in ServerPull

`ServiceAccessFilter` respectively. The collaboration use `USaf2` of `Checking-AccessRights` binds the `authorisee` and `authorisor` roles to the `Service-AccessFilter` and `AccessServer` roles respectively. Goals expressions are defined for each of the collaborations separately, as well as for the composition of these in `UserPull`. Dynamic linking of the interactions of the structural parts can be expressed using composition policies as explained in Section 5. above in the main body of this report.

D.3 Access control models

Although not shown in the authentication and authorization patterns presented in Appendix C and Appendix D, an access control model is needed to administer access rights (permissions) and enforce access control policies. In this section we briefly describe the well known access control models, focusing on the role-based access control model (RBAC) and then we explain the interfaces between a RBAC infrastructure and the `UserPull` and `ServerPull` architectures, respectively. A detailed overview of different access control models is given in [56].

Several models for access control have evolved such as discretionary access control (DAC), mandatory access control (MAC), and others [12]. Role-Based Access Control (RBAC) has emerged as a scalable alternative, and has been the focus area for recent research on access control resulting in numerous model variants. As explained above in Sect. 4.2 of this report, we assume that a RBAC model is used with the AA-patterns.

There are five administrative elements in the basic RBAC model: (1) Users, (2) Roles, and (3) Permissions. Permissions are composed of (4) operations applied to (5) objects. Fig. 52 shows the basic RBAC conceptual model. In RBAC, users are assigned to roles based on competencies, authority and responsibilities. Permissions, an abstract concept that refers to the arbitrary binding of computer operations and resource objects, are assigned to roles. A permission is a an approval of a particular mode of access to one or more objects in the system or some privilege to carry out specified actions [1]. An object is any protected resource. In RBAC, users are not directly granted permissions to perform operations on an individual basis. Instead, permissions are assigned to roles dynamically as an organisation changes and evolves. A user may establish several sessions simultaneously, and each session may have a different combination of active roles [12].

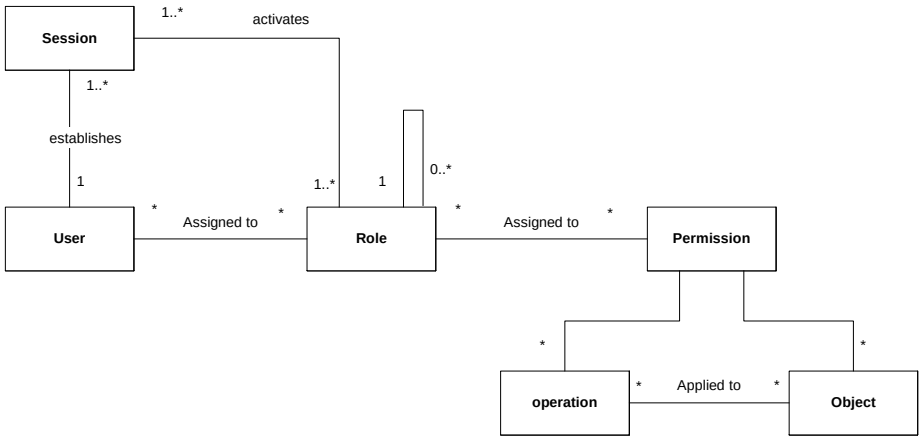

Figure 52: Role Based Access Control Model

The notions of subject and objects are used in RBAC relationships. A subject is an active entity in the system, e.g., a process or task that operates on behalf of the user within the computer environment. In our work on service composition in a service oriented architecture, we use the notion of a session between collaborating roles (service session) instead of subject as is common in the RBAC models.

There are two stages to acquiring permissions in a RBAC access control system. The first stage is authentication of the user (we have modelled this as the two party authentication pattern). The second stage is activation of roles. Once roles are acquired, permissions (access authorizations) may be specified for roles based on credentials, prerequisite roles, and policy. Authorization constraints are an important part of RBAC. Authorization constraints implement the access control policy for access to a service and include preconditions that must be satisfied before access to a service may be granted. The structure and form of the authorizations depends on on which RBAC model is to be deployed.

For applying the `UserPull` patterns, shown in Fig. 45, the instance playing the `AccessServer` role must have an interface to the RBAC infrastructure in order to obtain user authorizations which are distributed to the instance playing the `User` role when the `Auths activation` collaboration executes. The instance playing the `Service Access Filter` role interfaces with the RBAC infrastructure to obtain the access control policies that are to be enforced by the instance playing the `authorisor` role when the `Checking Access Rights` collaboration executes.

For applying the `ServerPull` patterns, shown in Fig. 49, the instance playing the `Access Server` role must have an interface to the RBAC infrastructure in order to obtain user authorizations status. The instance playing the `Access Server` role also interfaces with the RBAC infrastructure (not shown in our UML 2.0 collaborations) to obtain the access control policies that are to be enforced when the `Checking Access Rights` collaboration executes.