An Extensible Framework for Specifying and Reasoning About Complex Role-Based Access Control Models

Christopher Alm

Institute of IT-Security and Security Law, University of Passau
christopher.alm@uni-passau.de

Technical Report, Number MIP-0901
Department of Informatics and Mathematics
University of Passau, Germany
January 2009

An Extensible Framework for Specifying and Reasoning About Complex Role-Based Access Control Models *

Christopher Alm

University of Passau, Germany
christopher.alm@uni-passau.de

Abstract. To date, no methodical approach has been found to integrate multiple access control extensions and concepts proposed for RBAC in an access control model that deals with the complexity of such a model and still leaves the model open for further extensions. As we know from the case studies of our research project [1], bringing together various access control concepts such as separation of duty, workflow-related concepts, and context constraints is necessary in real world scenarios such as in the health care sector and in the financial sector.
To solve this problem, this report presents an extensible and flexible framework for the specification of complex RBAC models that is based on the modularization of access control concepts. Each concept is packed into a so-called *authorization module* and can then be reused and combined with other modules in order to specify a full access control model. The framework can be used to define new access control concepts rapidly and concisely as well as to explore and analyze them thoroughly. Furthermore, it is capable of delivering a policy data model for each generated access control model which can be used to develop an appropriate policy language.
As a method we use formal, object-oriented specification in the Object-Z notation. In particular, we demonstrate how formal reasoning can be applied in order to provide an in-depth analysis of the specification.

1 Introduction

A multitude of access control concepts and models based on the role-based access control (RBAC) paradigm addressing various access control requirements has been proposed. Examples are role hierarchies [5], context constraints [37], workflow related concepts [33, 11, 34], separation of duty [25, 5], obligation [36], and temporary delegation of rights [18, 51]. These access control concepts are specified as an extension to some notion of RBAC at the core by using a specification method that is appropriate for the particular purpose. The methods used range from informal descriptions including formulas and diagrams to declarative

* This work was supported by the German Ministry of Education and Research (BMBF) as part of the project ORKA, http://www.orka-projekt.de

programming languages such as Datalog and specification languages such as Z. While the chosen methods are particularly appropriate to make clear the idea of the newly proposed concepts, the situation can become complicated when it comes to reusing, combining, and integrating concepts from different models in order to specify them in a complex combined model. Hence, the problem is that we have many access control concepts for various scenarios but there seems to be no suitable method available to put them together that also handles complexity and still leaves the generated models open for further extensions.

As an example, consider the RBAC standard [5] striving for an integration of four concepts—namely roles, role hierarchy, and two variants of separation of duty. They are specified by using informal descriptions plus fragments of the Z specification language. We think that it will not be feasible to extend the framework presented in the standard by additional concepts because the way they are integrated has already reached its limits. We see the reason for this in that this way is error-prone and will easily lead to inconsistencies. Even though Li et al. [35] achieved a great deal of improvement, their approach is still based on the same methods and we think it does not solve the general problem.

Combined access control models incorporating various concepts originating from different models are necessary in real world scenarios such as in the financial sector or in the health care sector [1]. The construction of an authorization engine for such a setting needs to be based on an appropriate authorization model combining the required concepts. Furthermore, if a unified reference model is established such as the RBAC standard, it should be based on a solid formal method addressing extensibility, complexity handling, reusability, and flexibility.

To address these issues the goal of this report is to provide a flexible framework for specifying complex and extensible RBAC models in a unified way. In particular our contributions are:

- We have developed a framework for the specification of complex RBAC models (cf. Sect. 3 and 4). It is based on the modularization of the participating access control concepts. Each concept is packed into a so-called *authorization module* and can then be reused and combined with other modules in order to form an access control model. The framework is designed with the systematic approach of formal object-oriented specification using Object-Z [22]. We show how to use multiple inheritance in Object-Z in order to modularize access control concepts and explain the main features of the framework.

  Firstly, the framework can be applied in order to generate a software model for the implementation of an authorization engine supporting a certain combination of access control concepts. Secondly, with the framework new concepts can be defined and explored rapidly and concisely. Finally, the framework as a whole can be used to establish a reference model for RBAC including various access control concepts which could not be done before in this way.
- We demonstrate how to apply formal reasoning in order to provide an in-depth analysis of the authorization modules (cf. Sect. 6). Thereby we can, for

example, prove important security properties as well as argue about design decisions.

- We are currently developing a set of authorization modules. A recent version is available online [7]. We already support role hierarchy, separation of duty, context constraints, Chinese wall, and workflow-related constraints including history-based separation of duty, prerequisites, binding of duty, and cardinality constraints. In this report we show excerpts from the specified modules in order to demonstrate the functionality and the capabilities of the framework. It should be noted that these modules also contain innovative aspects, which will be discussed as future work.
  The current set of modules builds the foundation of the authorization model of the ORKA research project [1]. The goal of ORKA is to develop a flexible and extensible authorization architecture that is able to enforce a wide range of organizational control principles and access control concepts.
- We have incorporated a functionality that each authorization module specifies which part of it is supposed to be included in the policy (cf. Sect. 5). Thereby the framework is able to deliver an abstract version of a policy language (i.e. a policy data model) for each generated access control model. We show how to use instantiation and polymorphism in Object-Z for this purpose. This feature is particularly useful for the definition of the policy language or policy representation format used by an authorization engine. Therefore, the development of the ORKA policy language (OPL) is based on our framework and XML [1].

This report is organized as follows. Section 2 gives more information on the motivation of a framework for combining access control concepts and explains the rationale for the method we used. While Section 3 introduces our idea from a high-level perspective, Sections 4 and 5 provide the necessary details. In Section 6 we demonstrate how to reason about the specification and discuss how we can benefit from such an analysis. Section 7 compares our solution with related work, Section 8 states our ideas for further work, and Section 9 concludes the report.

2 Motivation and Method

2.1 Combining Access Control Concepts

Bringing together various access control concepts such as separation of duty, workflow-related concepts, and context constraints is necessary in order to establish a unified reference model for RBAC as well as for the construction of an authorization engine supporting a certain combination of such principles. From the results of the case studies conducted as part of our research project [1], we know that various combinations of concepts are particularly required in real world scenarios, for example, in the health care sector and in the financial sector.

The ANSI RBAC standard is an example that combines four concepts: roles, role hierarchy, and two variants of separation of duty. Their integration is entirely based on informal textual descriptions without using a dedicated method.

By using the *ad-hoc* method instead, the standard is on the one hand at the limit of being able to handle the complexity of the integration of the four concepts. On the other hand, it would be tedious and cumbersome to extend it by further concepts while reusing the existing definitions. In our opinion, textual descriptions are error-prone and should not have a normative character. Instead they should be replaced by a solid, dedicated specification method and serve only for illustrative purposes.

Another prominent example is the access control model of XACML, which is introduced by informal descriptions and diagrams [36]. The drawbacks became evident when XACML was unable to be extended to support separation of duty [8]. To date, no proper solution has been found, yet. Current solutions such as Crampton's approach [17] based on blacklists and XACML's obligation mechanism do not solve the problem fundamentally, but rather are developed as a patch work. We see the reason for this in that the underlying access control model of XACML was not designed to be extensible and was not defined by using a dedicated method suitable to achieve extensibility. Note that XACML's built-in extensibility is a powerful concept that can involve almost arbitrary context information into the process of access decision making, but it cannot be used to implement separation of duty properly.

In general, we see the key characteristics of access control models integrating multiple access control concepts as follows:

- They are complex: while integrating multiple concepts in one model the number of formulas and definitions increases. These definitions and formulas may interact or rely on each other and thus they may overlap or even conflict with each other. The same is true for textual descriptions. Already the ANSI standard is a fairly complex model with many formulas and definitions for four basic concepts.
- They need to be open for extensions and change: once new access control requirements arise, for example, due to the arrival of new technologies, it will be necessary to adapt an access control model accordingly. We think that the trend of newly emerging and changing requirements, as recognized by Botha [14], still continues.
- They need to be precise: such a model should not leave room for ambiguities in order to avoid misinterpretation and hidden design flaws.

These issues are not addressed by ad-hoc combination solutions. As a consequence, ad-hoc textual combination is not sufficient to specify complex RBAC models including various access control concepts and principles. They need to be replaced by a systematic approach addressing the mentioned drawbacks.

2.2 Policy Data Modeling

A policy language is a common vehicle to make the policies that are run by some authorization engine (which in turn is based on some access control model) manageable with regard to administration and enforcement. Therefore, it may

be necessary to develop a policy language for an access control model which is able to represent the policies of the model. There are, for example, approaches adding such a representation format for policies to an RBAC-based model [27, 12, 10].

For this reason, we incorporate data modeling facilities for a policy language. Having these facilities, it is not necessary to develop a policy language from scratch each time a model generated by the framework is implemented. Instead, the data model for policies, which is generated in conjunction, can be used as an abstract version of a concrete policy language.

Note that this approach of developing a policy language is driven by the underlying access control model. There are also *language-driven* approaches where the development of the access control model is driven by the syntax of a policy language [36, 19].

As part of the ORKA research project [1], a policy language on the basis of our framework and XML is currently under development. For this language the flexibility and extensibility of the framework is inevitable because ORKA aims to incorporate access control principles for various application scenarios including banking and health care. Furthermore, it strives for being open for extensions so that it can still be used when the requirements and the application environment change.

2.3 Rationale for the Method

This subsection gives the reasons for choosing formal object-oriented specification using Object-Z as the method to specify our framework. In particular, we draw conclusions from the preceding subsection.

Specification Language Specification languages such as Z [49], Object-Z [46], Alloy [30], VDM [31], and UML are designed for the specification of software systems. Therefore, they are highly suitable to express the structures of software systems such as their operations and state spaces. In particular, state-based modeling is an important feature that is not easy to achieve with textual methods, non-modal logics, or mathematical text with formulas (e.g. state functions and state transitions need to be introduced manually).

Specification languages provide a systematic and methodical approach for the specification of systems and therefore help to reduce inaccuracies and errors. Furthermore, they can usually be integrated systematically in the whole software development process.

In our experience, reading and writing specifications in a specification language is much faster and clearer than with textual specifications—once acquainted with the language.

Finally, many specification languages offer tool support so that specifications can be checked and analyzed (semi)-automatically [2, 41].

Object Orientation As has been argued elsewhere [13, 48], object-orientation turns out to be a modeling paradigm particularly suitable to produce flexible and extensible models of complex systems. Among its key features to address complexity there are generalization (to structure the system in an inheritance hierarchy and to organize and share common attributes and services) and data encapsulation (to provide information hiding, abstraction, and system decomposition). Modularity and feature reuse contribute to achieve extensibility and flexibility. In particular, object-orientation is also suitable for modeling the data of a system.

Therefore, with respect to our goals we consider object-orientation to be the appropriate basis for our framework.

Formal Method There has always been a big debate about the benefits and drawbacks of formal methods [15]. We do not want to raise the discussion again because this would be well beyond the scope of this report. However, we point out the key issues why we think a formal method is the appropriate vehicle in order to specify an access control model:

- Formal methods are precise and leave no room for ambiguities. Thereby they can avoid misinterpretation and can protect from hidden design flaws.
- In addition, formal specifications can be subject to formal reasoning. Thereby certain properties of the specification such as important security requirements can be proven formally.

We think that particularly in the area of security critical software such as an authorization engine, the application of formal methods is appropriate at design time. Finally, it is important to note that for the specification and administration of policies at runtime no formal method is necessary in our approach. Hence, administrators do not have to worry about this (cf. Section 5.2).

Object-Z We have chosen the Object-Z notation [22, 46], which is derived from the Z notation [49], for the specification of our framework. Object-Z is a mature specification language with a well-understood semantics that is both formal as well as object-oriented. As a consequence, Object-Z fits the requirements stated in Sections 2.3, 2.3 and 2.3.

Object-Z can be used to create self-contained formal specifications that are clearly separated from the informal text. Textual descriptions or diagrams (e.g. ER-diagrams showing the state space of a class) can still be used for illustration, which is recommendable. The class construct of Object-Z provides the formal structuring capabilities necessary for this. In particular, the suitability of Object-Z for the definition of standards and reference models has been proposed [23]. Besides being a formal language, as a main reason these structuring capabilities are given: the Object-Z class construct and the way operations can be expressed offers the succinct specification of the various hierarchical relationships and the communication between objects in a large and complex system, which standards typically are [23, p.1].

The ability of our framework to express access control concepts is limited by the expressiveness of the chosen specification language. Object-Z is widely accepted as a highly expressive formalism which is based on typed set theory and first-order logic. Currently we are not aware of any access control concept that is not expressible in Object-Z.

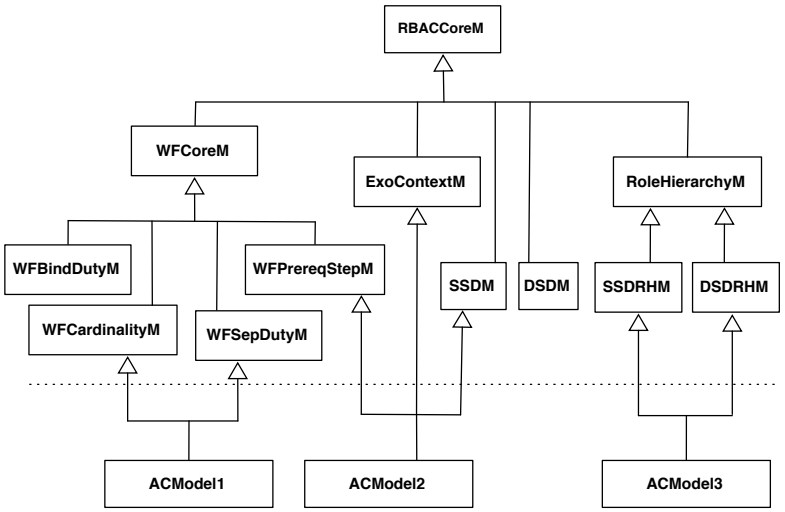

Fig. 1. Class Inheritance Hierarchy (UML)

Like Z, Object-Z was designed to be readable by humans which is based on a rich syntax including common mathematical notation displayed by means of LaTeX. We see this as one of its main advantages over other notations, particularly for the definition of reference models. However, the fact that is readable by humans rather than by computers is also a reason for one of its drawbacks: the only tool currently available supporting the analysis of Object-Z specifications is Wizard and the functionality of this tool is limited to type checking [3].

At the cost of human readability, this issue is addressed by the ASCII-based Alloy language and its analyzer tool [2]. Alloy is a lightweight formal specification language that has also a notion of object-orientation [30]. Even though one could argue that human readability already outweighs the availability of an analyzer tool, there is another reason why using Alloy for the definition of our framework is not possible. Alloy does not support multiple inheritance which is one of the key ideas behind our framework [29].

Finally, note that semantic details of Object-Z are introduced when they are necessary for the description of the framework in Sections 4 and 5. For convenience we added a list of used symbols as a quick reference in Appendix A.

3 Framework

In order to solve the problems described in Sections 2.1 and 2.2, the idea of this report is to define a framework consisting of so-called *authorization modules*. Each modeled access control concept (such as separation of duty) is packed into an authorization module (cf. Section 3.1). Afterwards, these modules can be reused and combined with each other in order to extend the framework (cf. Section 3.2) or to generate full access control models (cf. Section 3.3). Finally, Section 3.4 explains on a high level how a policy data model is included in each generated model.

3.1 Module Definition

An authorization module is introduced by means of one Object-Z class, which is by convention of our framework a class ending with the letter "M". Such a class specifies an access control concept including the access decision evaluation logic, the administrative interface, the interface to the system that raises access requests, and the state variables for both policy information and dynamic context information.

3.2 Framework Extension

Each time a new authorization module is introduced to the framework it can reuse the existing modules by inheriting their features. Thereby a module can add or change some of the functionality of another module while leaving and just reusing the rest as it is. This is realized through class inheritance in Object-Z. Modules independent from each other stay clearly isolated and removing or altering one of them does not affect the other modules. Note that, as a consequence, the dependences of access control concepts is clearly specified in our framework.

Above the dotted line, Figure 1 shows our current set of authorization modules and how they depend on each other[1]. For example, *RBACCoreM* comprises basically the core component of the RBAC standard, *RoleHierarchyM* adds a role hierarchy, *SSDM* and *DSDM* add separation of duty, *ExoContextM* adds exogenous context constraints similar to Strembeck and Neumann's approach [37], and WFCoreM and its submodules add workflow related concepts such as history-based separation of duty and prerequisite steps. The formal definition of these modules is available online [7].

There are three remarks worth mentioning. Firstly, even though the framework is designed in such a way that it needs to have a root element, this does not need to be *RBACCoreM*. A more general root element could be introduced, so that access control concepts without a notion of roles can be added. Secondly,

[1] Note that we visualize Object-Z class relationships by means of UML class diagrams which only have an illustrative purpose. They are not necessary for the definition of the framework.

it becomes necessary to use Object-Z's multiple inheritance for the definition of modules. when an access control concepts depends on more than one existing concept. As an example, consider a workflow scenario where a history-based separation of duty constraint is only applied if some context condition is true such as an amount of a loan exceeding a certain amount in a loan origination process.

Finally, it is important to note that the choice of authorization modules is depending on many design decisions. Thus, it is well beyond the scope of this report to discuss the questions of which modules to use for the framework and how these modules are designed internally.

3.3 Generating Access Control Models

An access control model can be generated by inheriting from all authorization modules corresponding to the required access control concepts. This is realized by Object-Z's multiple inheritance allowing us to merge the state spaces and operations of different modules and therefore to include the functionality they provide. When selecting a module, all its parent modules are automatically included. Below the dotted line of Figure 1, three examples of access control models are given—namely $ACModel1$ to $ACModel3$. Note that these are not part of the framework. Thereby the framework provides a formal way to express recommended combinations or guidelines for combinations.

Finally, it should be noted that the generation of an access control model differs from the extension of the framework by an authorization module in such a way that a model does not add or alter functionality, on the one hand. On the other hand, a generated model may set some constants occurring in the definition of a participating module in order to provide the actual parameters for this participating module.

3.4 Generating Policy Data Models

For each authorization module there is exactly one associated Object-Z class which has by convention the same name except it is ending with the word "Policy" instead of "M". The purpose of these Object-Z classes is to provide the data model for a policy language. Hence, these *policy modules* define which information is supposed to be stored within a policy for each authorization module. The policy modules are arranged in the same class inheritance relation as the authorization modules. When generating an access control model, the set of policy modules corresponding to the selected authorization modules generate the appropriate *policy data model*.

Figure 2 illustrates this process by showing the dependency of three exemplary authorization modules generating an access control model and the associated policy modules as well as the policy data model. In this way, a concrete policy language (e.g. as an XML application) can be developed for an access control model on the basis of the automatically delivered policy data model.

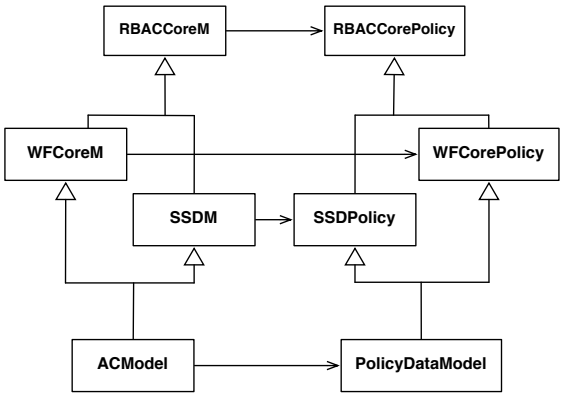

Fig. 2. Dependancy between Authorization and Policy Modules (UML)

4 Authorization Modules

While Section 3 introduced our framework from a high level perspective, this section provides the details by going through the features of a module step by step.

4.1 Overview

Figures 4 and 3 show an abridged version of *RBACCoreM*, which is derived from the core component of the RBAC standard. The parts cut out are displayed by "...". Each module definition starts by its visibility list indicated by the symbol "⌈". The visibility list is the set of operations and state space variables that can be accessed from outside. Thereby the whole interface of a module is defined. Note that we currently do not model any review functions.

The first box—in Z parlance called *schema*—contains the state space of the module comprising the set of state variables in the part above the line and the class invariants below the line. The class invariants are predicates that need to be fulfilled at all time in the life cycle of a runtime instance based on this module. In case of *RBACCoreM* the state variables include, for example, a set of subjects, which represent the active entities of the system, and a policy, which contains information such as the set of roles and the role assignment relations[2]. The class invariant ensures that a user can only activate a role (on behalf of a subject) if he or she is assigned to it.

In the remainder all operations of the module (whether publicly visible or invisible) are defined. An operation can be defined in two ways: either by means

[2] Policies are introduced in Sect. 5.

of a schema as in the case of *ImportPolicy* or by means of the schema calculus as in the case of *CheckAccess*:

- In the former case, an operation schema contains the communication variables of the operations in the part above the line and the pre- and post-conditions for the operation in the part below the line. The communication variables are decorated either with "?" to indicate an input variable or with "!" to indicate an output variable. The state of the variables in the Δ-list is supposed to be changed by the operation. By the variable decoration "′" one can refer to the state of a variable after the change.
- In the latter case, existing operations can be promoted by using the operation composition operators of the Object-Z schema calculus. This is an elegant way to reuse and modify features from parent modules.

The visible operations of an authorization module such as *CheckAccess* are typically the ones that are invoked by the system component that raises the access requests such as a policy enforcement point. Administrative operations that are used, for example, to add a user are defined within the policy modules (cf. Sect. 5).

All in all, an authorization module can be conceived as a set of authorization constraints (predicates: class invariants, post-, and pre-conditions) posed on state variables and operations in order to model a certain access control concept.

Note that the state variables and the operations of a module constitute the *features* of a module.

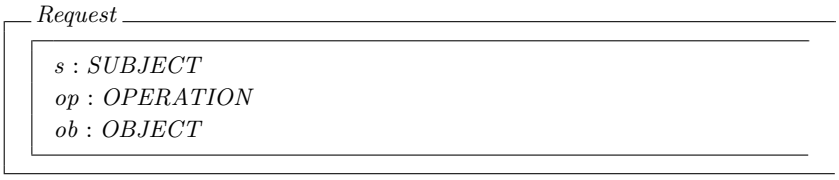

Fig. 3. Basic Request (Object-Z)

4.2 Data Encapsulation

For the internal specification of a module, data encapsulation is a powerful means to handle complexity as well as to provide flexibility. In Object-Z this is realized with types: all declared variables in Object-Z need to be assigned a type. E.g.

$$U : \mathbb{P}\, USER$$

declares that the variable U has the type power set of *USER* (i.e. U is an element of $\mathbb{P}\, USER$, hence, U is a set of elements of the type *USER*). Types

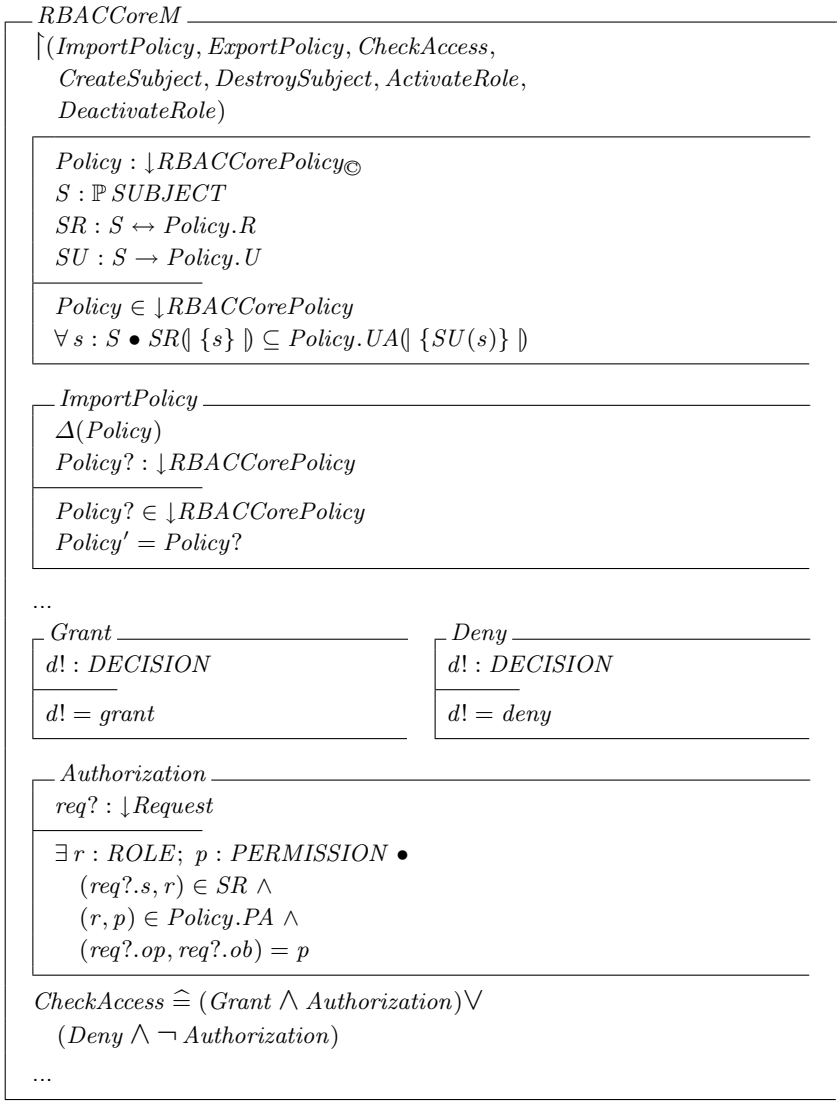

Fig. 4. RBAC Core Module (Object-Z)

can be constructed on top of so-called *basic types* or *object types* by using type constructors such as "ℙ" (power set) and "×" (cartesian product).

– Basic type: a basic type is just defined by introducing the name of the type such as

$$[USER]$$

Thereby all the details of the type are up to the interpretation of the implementation. In the case of *USER*, the elements of this type can be interpreted as identifiers to the digital representation of human users of the system, as they are provided by, for example, a directory service. Hence, *USER* can be conceived as a namespace for system users.
It is also possible to introduce basic types by listing all its values such as for the possible access decisions:

$$DECISION ::= grant \mid deny$$

– Object type: in contrast to basic types, an object type models (and particularly hides) all the details about the type. This information can then be used by the access decision logic in order to make access decisions depend on it. For example, Figure 5 shows our model of a workflow management system (WFMS) as it is seen by the access control system. This WFMS is made available to the access control model in the *WFCoreM* module by just introducing a state variable of the type *WFMS*

$$wfms : WFMS$$

Thereby all the information internal to the WFMS is hidden behind one variable, achieving a clear separation between the authorization engine and the WFMS in terms of the modeling. In order to model history-based separation of duty, for example, our *WFSepDutyM* can access the history through $wfms.History_{WFI}$ [7].
Through this principle of information hiding, object types are a way to cope with the complexity of multiple access control concepts and with the variety of information these concepts deal with.

Furthermore, for the extension of our framework by further modules as well as for the restructuring of an existing module internally, this type system makes an important contribution for the flexibility of the framework.

For example, in order to implement obligations in the sense of XACML, the basic type *DECISION* could be replaced by an appropriate object type incorporating obligations. Afterwards, a module *ObligationM* could be added assigning obligations to permissions and extending the *Grant* and *Deny* operations accordingly.

Another example is to extend existing object types such as *WFMS* by means of inheritance in order to include further information such as the business logic of processes. A submodule *WFBusinessLogicM* could then extend *WFCoreM* in order to enforce order of events by relying on the extended version of *WFMS*. Note that currently our workflow modules only enforce security requirements such as history-based separation of duty.

4.3 Inheritance

The framework can be extended with a further access control concept by adding an authorization module that models this concept. This is realized by Object-Z's

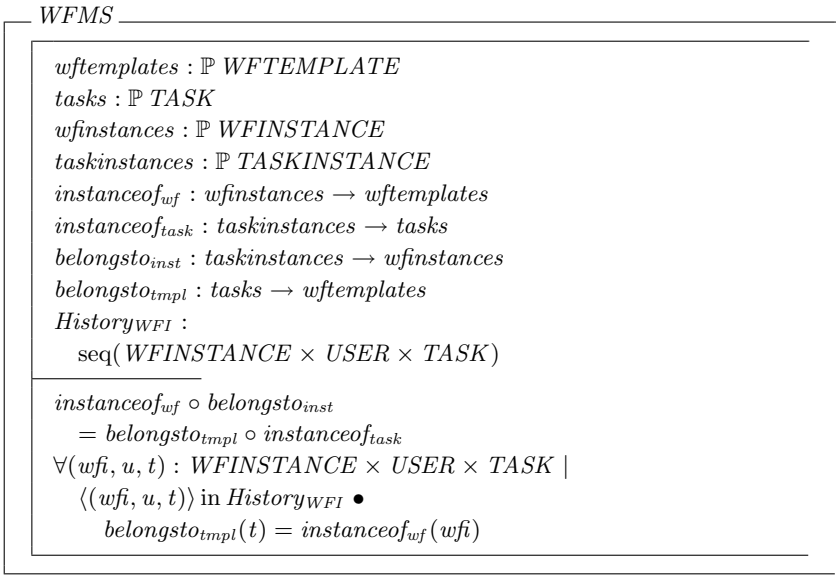

Fig. 5. A Workflow Management System Model (Object-Z)

elaborated class inheritance capabilities and thus an added authorization module can reuse the features of one or more existing modules in a fine-grained manner. Figures 7 and 6 show the *WFCoreM* module introducing our core module for workflow related constraints as a submodule of *RBACCoreM*. This module adds an additional authorization at task layer (via *ClaimTI*) and ensures that requests at object layer (via *CheckAccess*) are in line with the task layer. Within our framework we make use of the following three cases to inherit a feature of a module (recall the definition of feature in Section 4.1):

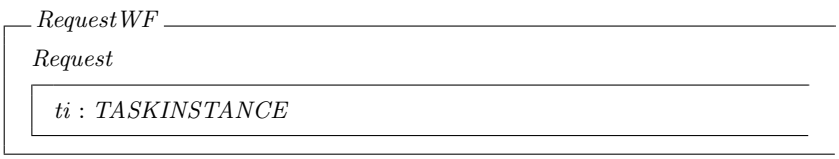

Fig. 6. Workflow Requests (Object-Z)

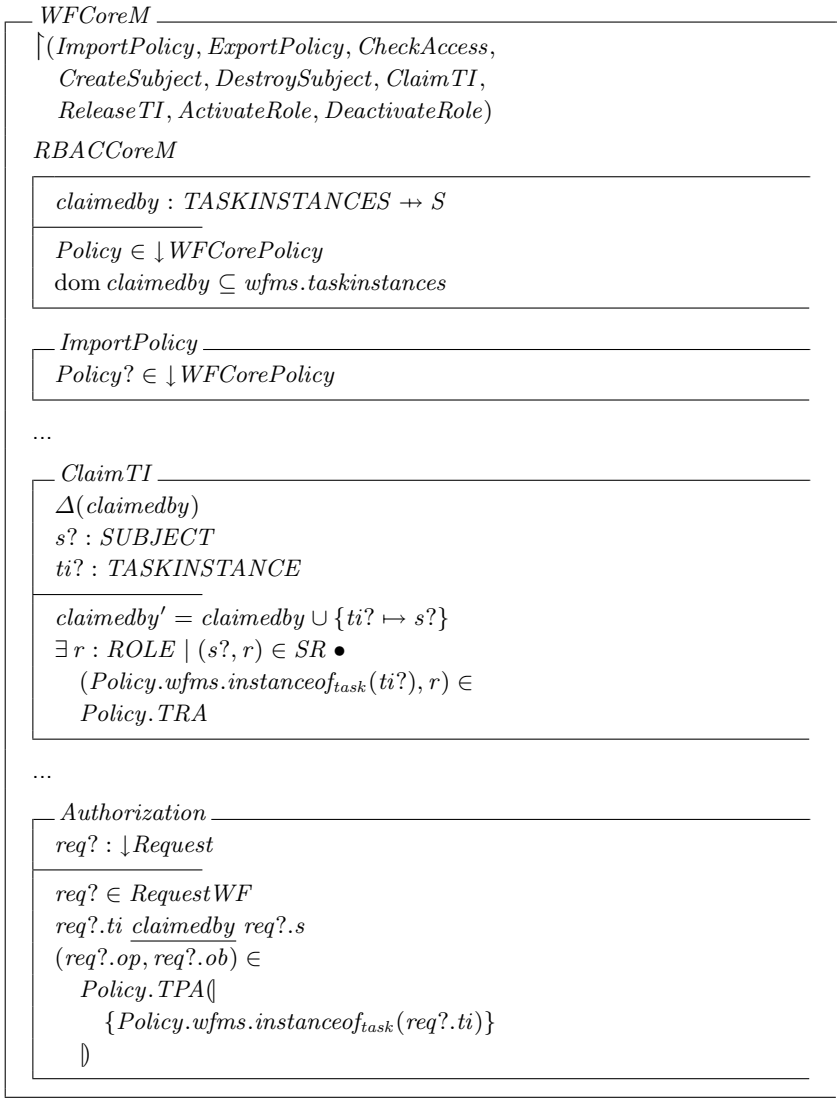

Fig. 7. Workflow Authorization Core (Object-Z)

1. Simple reuse: if the submodule does not specify anything about a feature, it is reused as it is. For example, the *DeactivateRole* operation or the state variable S is reused by *WFCoreM* as it is.
2. Merge: if an operation with the same name and same type signature is defined, the predicates of this operation stated by the submodule are implicitly conjoined (i.e. added with logical "and") with the predicates defined (and

thus inherited) in the parent module. An example is the *ImportPolicy* operation of *WFCoreM*. In case of *Authorization* two further constraints are added to the operation originally defined in *RBACCoreM*.
The actual state space of a submodule is the union set of the (inherited) state variables of the parent module and the set of state variables defined explicitly in the submodule. Hence, the set of state variables of the parent module is implicitly contained in the state space of the submodule. All class invariants (inherited from parent module or explicitly stated in submodule) are conjoined. In the case of *WFCoreM*, the function *claimedby* is added which tracks the task instances claimed by (i.e. assigned to) a subject. A corresponding class invariant is added ensuring that only task instances known to the WFMS are claimed.

3. Cancel and redefine: a submodule can also cancel a feature of a parent module in order to redefine it with a whole new semantics. We use this principle, for example, to introduce role hierarchies [7].

Thus, by means of inheritance, further constraints (predicates: class invariants, pre- and post-conditions) can be posed on the existing state variables and operations of a module. Also, new constrained state variables and operations can be introduced as well as existing constraints can be canceled.

4.4 Module Combination

The combination or selection of modules for the definition of an access control model is a special case of inheritance. By using multiple inheritance, all desired modules can be included while merging their constraints and state variables according to the Object-Z inheritance semantics as described in Section 4.3: state variables are merged, predicates are conjoined. Thus, all the constraints are collected and placed automatically to the right places. Note that commonly known issues of multiple inheritance due to name clashes or inheritance of unnecessary attributes, are problematic in object-oriented programming rather than in object-oriented modeling [48].

Figure 8 shows an example of a model including static separation of duty, exogenous context constraints, and the prerequisite step principle for workflows. The cut out visibility list needs to be stated explicitly. It is the union set of all visibility lists of the participating authorization modules and defines the interface to whole the access control model. *ExportPolicy* has been cut out as well.

Now, *ACModel*2 can be used as a software model integrated in the development process of an appropriate authorization engine.

5 Policy Representation

5.1 Policy Modules

As described in Section 3.4 on a high level, authorization modules are associated with policy modules in such a way that each authorization module instantiates its corresponding policy module. Operations to import and export

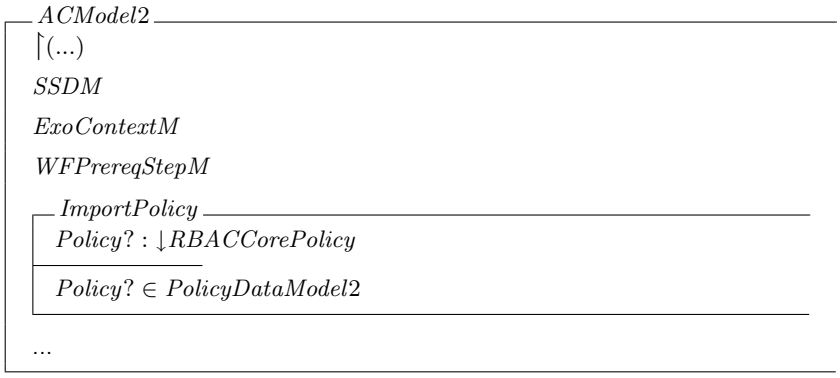

Fig. 8. Access Control Model 2 (Object-Z)

instances of policy modules are available in each authorization module. Figure 9 shows the policy module *RBACCorePolicy* and *WFCorePolicy* which correspond to *RBACCoreM* and *WFCoreM*, respectively. They contain everything that is supposed to be included in a persistent policy for *RBACCoreM* and *WFCoreM*—namely users, roles, permissions, user role assignment, and permission role assignment for *RBACCoreM*, and task permission assignment, task role assignment, and a link to the WFMS for *WFCoreM*. In contrast, subjects and their activation relations, for example, are not supposed to be part of a policy. Policy modules also define administrative operations such as for adding and deleting users, which are usually invoked by a policy administration point. These have been cut out here. The class invariants of the policy modules define *administrative constraints* ensuring that static requirements of the policy are not violated during the policy administration process. An example is static separation of duty, where the policy module ensures that no user is assigned to two critical roles.

ImportPolicy and *ExportPolicy* operations are responsible for loading and exporting policy information in an authorization module. Figures 4 and 7 show the specification of *ImportPolicy* (*ExportPolicy* is analogous). The communication interface of *ImportPolicy* makes use of polymorphism in order to have the same type signature throughout the whole inheritance hierarchy (i.e. signature compatibility). As shown in Figure 7, each authorization module adds a constraint to *ImportPolicy* (here $Policy? \in \downarrow WFCorePolicy$) in such a way that the loaded policy needs to be an instance of the appropriate corresponding policy module (or one of its submodules). Thereby, it is always ensured that an imported or exported policy module instance has the data format appropriate for an authorization module. I.e. at least the information needed by the authorization module is included in the loaded policy.

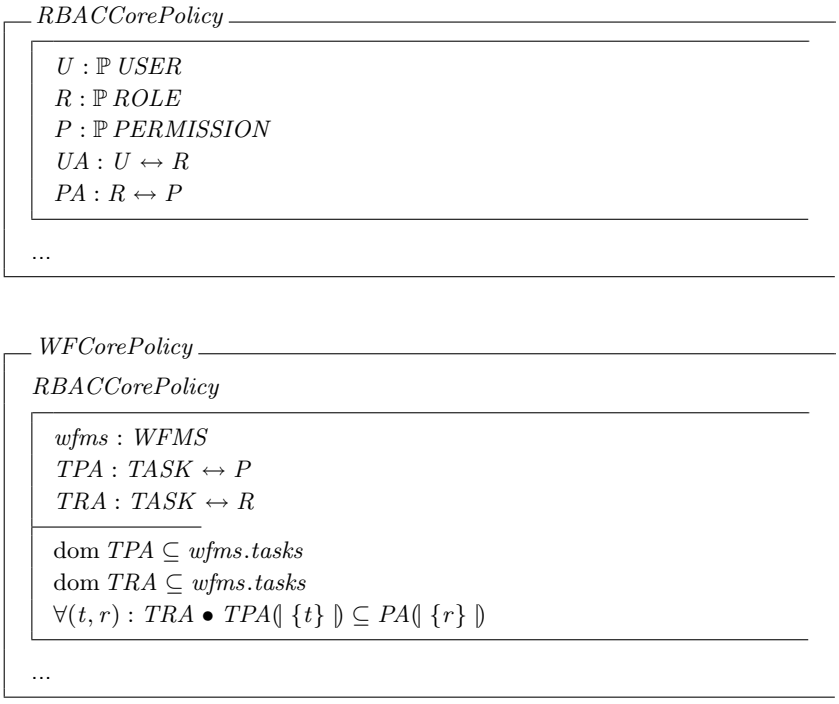

Fig. 9. RBAC and Workflow Core Policy Modules (Object-Z)

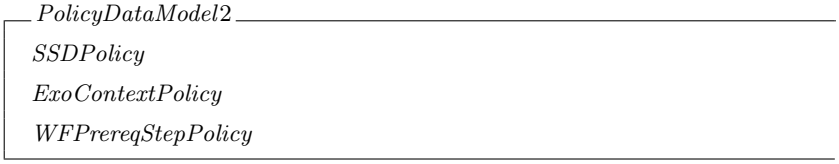

Fig. 10. Policy Data Model 2 (Object-Z)

5.2 Policy Data Model

The generation of a policy data model corresponding to an access control model is analogous to the generation of an access control model. It makes use of multiple inheritance so that the appropriate data models from all the necessary policy modules are included. I.e. state variables are merged and class invariants (if any) are conjoined.

Figure 10 shows *PolicyDataModel*2 which is the policy data model corresponding to *ACModel*2. Hence, instances of this policy data model Object-Z class are the policies (at modeling level) of the access control model *ACModel*2.

For an authorization engine implementing *ACModel*2 it would be necessary to define a concrete policy representation format, for example, as an XML application (as in ORKA [1]) or by means of data definition language such as SQL. Also graphical representations that are particularly suitable for policy administration are possible. The formal semantics of the policies is elegantly defined through *ACModel*2 in any case.

By using an ad-hoc concrete syntax for *PolicyDataModel*2 (i.e. names imply the relationship to the data model), an example policy for *ACModel*2 could be stated as follows. Imagine a scenario with the following users, roles, permissions, user assignments, and permission assignments:

```
users         A, B, C;

roles         clerk, supervisor, manager;

permissions   readdoc, writedoc, signdoc,
              createdoc, syscleanup;

A   user-assigned-to   manager;
B   user-assigned-to   clerk;
C   user-assigned-to   supervisor;

clerk      assigned-to-permission  createdoc;
clerk      assigned-to-permission  readdoc;
clerk      assigned-to-permission  writedoc;
manager    assigned-to-permission  signdoc;
manager    assigned-to-permission  readdoc;
supervisor assigned-to-permission  syscleanup;
supervisor assigned-to-permission  readdoc;
```

Two static separation of duty constraints are placed. The "1" indicates that at most one role may be assigned to a user.

```
critical-roleset(1) { supervisor , clerk } ;
critical-roleset(1) { supervisor , manager } ;
```

Roles are assigned to the tasks `apply`, `check`, `review`, and `approve`. Afterwards, the tasks are assigned to permissions appropriately. A special prerequisite condition enforces that the task `review` must be completed before `approve` can take place:

```
wfms wfms.mydomain.org;

apply    task-assigned-to-role  clerk;
check    task-assigned-to-role  clerk;
review   task-assigned-to-role  supervisor;
approve  task-assigned-to-role  manager;
```

```
apply    task-assigned-to-perm  createdoc;
check    task-assigned-to-perm  readdoc;
check    task-assigned-to-perm  writedoc;
review   task-assigned-to-perm  readdoc;
approve  task-assigned-to-perm  readdoc;
approve  task-assigned-to-perm  signdoc;

review   must-be-completed-before  approve;
```

Finally, a context constraint is placed in such a way that system cleanup can only happen on Saturdays:

```
cc1 { timesrv.mydomain.org , DayEquals,
      "Saturday" };
syscleanup  permission-assigned-to-cc  cc1;
```

Here, `cc1` is the identifier for the context constraint.

6 Reasoning About the Specification

In this section we demonstrate by example how we can benefit from the reasoning techniques available for the Object-Z specification language. The style in which we present the proofs has been used successfully elsewhere [16]. We think that such hand-made proofs are easy to read and understand and that they are appropriate to proof typical system requirements and properties. By using a logic with inference rules such as $\mathcal{Z}_C$ [26] and $\mathcal{W}$ [45], the proofs can be adapted to be checked by a theorem prover. As a future work, we may consider this direction.

6.1 Verifying Properties

A class invariant specifies explicitly a property of a class that must hold all the time. The class invariants are chosen in such a way that they best clarify the functionality and intent of the class [22, p.95]. Therefore it may be desirable to leave out class invariants that follow from other class invariants, type definitions, or operation definitions. However, these so-called *derived invariants* may contain a valuable insight into the specification from a certain perspective (such as important high-level security requirements) so that one may want to state them separately and then to prove that they indeed follow from the specification.

For example, in the version of role-based access control published in 1992 [24] there was a rule called *role assignment rule* stated as follows "a subject can execute a transaction only if the subject has selected or been assigned to a role". This rule has been silently dropped in later versions of RBAC. We show that this rule is indeed redundant, i.e. that it is a derived invariant of the specification.

Proposition 1. *For any* $s : S$ *and* $(op, ob) : Policy.P$ *within the scope of RBACCoreM such that*

$$Authorization[s/req?.s, op/req?.op, ob/req?.ob]$$

it follows that

$$SR(\!| \{s\} |\!) \neq \varnothing$$

Proof: Assume that there exist $s : S$ and $(op, ob) : Policy.P$ such that

$$Authorization[s/req?.s, op/req?.op, ob/req?.ob]$$

but also $SR(\!| \{s\} |\!) = \varnothing$. However, from

$$Authorization[s/req?.s, op/req?.op, ob/req?.ob]$$

it follows that there exists $r : R$ such that $(s, r) \in SR$ which yields a contradiction. □

Another example is a requirement for workflows that is implicitly fulfilled by our specification (cf. Fig. 7 and 9). It can be stated as follows.

Proposition 2. *A subject who claimed a task instance has sufficient permissions in order to execute all operations associated (over Policy.TPA) with the corresponding task.*

Proof: Let $s : S$ and $ti : TASKINSTANCE$ such that ti $\underline{claimedby}$ s. Let the corresponding task be $task : TASK \mid task = Policy.wfms.\overline{instanceof}_{task}(ti)$. We need to show that s has the authorization to perform any permission $(op, ob) = p : PERMISSION$ that is assigned to $task$, i.e. $(task, p) \in Policy.TPA$. Hence, we need to show

$$Authorization[s/req?.s, op/req?.op, ob/req?.ob]$$

The only way to claim a taskinstance (see Δ-list) is by means of $ClaimTI$. Therefore, we have

$$ClaimTI[s/s?, ti/ti?]$$

from which we can follow that there exists a role $r : ROLE$ that has been activated by s, i.e. $(s, r) \in SR$, and that is assigned to $task$, i.e. $(task, r) \in Policy.TRA$. The class invariant of $WFCoreM$ ensures that for every such assignment $(task, r) \in Policy.TRA$ it holds that if $(task, p) \in Policy.TPA$ then $(r, p) \in PA$. Hence, we have $(s, r) \in SR$ and $(r, p) \in PA$ which concludes the proof. □

6.2 Consistency Analysis

An important requirement is the consistency of an access control model generated by the framework.

Definition 1 (Consistency). *An access control model is called consistent if there exists a state fulfilling all class invariants.*

As a consequence, such a model does not have any contradictory constraints. This is usually achieved by showing the *initialization theorem* for the class constituting the access control model, i.e. showing that an initial state exists [52]. Hence, if $State : Exp$ denotes the state schema of a class such as $ACModel2$ and $StateInit$ its initial state, we need to show that

$$\exists\, State' \bullet StateInit$$

Since we have currently no initial states defined in our framework, we need to defer an explicit example for future work. As Woodcock and Davies point out [52], an initialization theorem can usually be proven by eliminating the quantified variables, since the initial state is most often defined with a number of equations.

6.3 Argue about Design Decisions

Another application of formal proofs is to find arguments for design decisions. If a solution is chosen, the reason might be that another solution will entail some undesirable consequence. Hence, a good argument for the decision is to provide a proof for this implication.

For example, we chose

$$\forall (t, r) : TRA \bullet TPA(\!|\, \{t\} \,|\!) \subseteq PA(\!|\, \{r\} \,|\!)$$

as a class invariant of *WFCorePolicy* instead of

$$\forall (t, r) : TRA \bullet TPA(\!|\, \{t\} \,|\!) = PA(\!|\, \{r\} \,|\!)$$

because of the following undesirable consequence:

Proposition 3. *If*

$$\forall (t, r) : TRA \bullet TPA(\!|\, \{t\} \,|\!) = PA(\!|\, \{r\} \,|\!)$$

then one role $: ROLE$ *cannot be assigned to two tasks* $t_1, t_2 : TASK$ *with different assigned permissions (over TPA).*

Proof: Let $p : PERMISSION$ be w.l.o.g. such that $(t_1, p) \in TPA$ but $(t_2, p) \notin TPA$. Assume that $(t_1, role) \in TRA$ and $(t_2, role) \in TRA$. By using $(t_1, p) \in TPA$ and $(t_1, role) \in TRA$ as well as the proposed formula we get

$$p \in PA(\!|\, \{role\} \,|\!)$$

By using $(t_2, role) \in TRA$ together with the proposed formula we get

$$PA(\!|\, \{role\} \,|\!) = TPA(\!|\, \{t_2\} \,|\!)$$

Hence, $p \in TPA(\!|\, \{t_2\} \,|\!)$, i.e. $(t_2, p) \in TPA$, which yields a contradiction. □

7 Related Work

A methodical approach by Ahn and Hu [6] is based on model driven development of RBAC models specified in UML/OCL. While the focus of our approach is mainly on modeling and analysis, Ahn and Hu additionally take into account methodical translation to enforcement code. The approach is also object-oriented, however, no particular extensibility issues or issues dealing with combination of RBAC extensions are addressed. While our approach to validation is proof-based, Ahn and Hu's model-based approach can only address consistency questions[3].

Schaad and Moffett's approach [44] based on the Alloy lightweight specification language and the Alloy analyzer tool deals with the specification, integration, and analysis of two RBAC extensions—namely separation of duty and the administrative model ARBAC97 [42]. A major benefit here is that models and specifications can be analyzed automatically with the Alloy analyzer to find contradictory constraints and thus to strengthen the specification (i.e. model-based consistency analysis). In contrast to our approach, Schaad and Moffett's approach does not attempt to be open for further RBAC extensions. As pointed out in Section 2.3, we see the lack of human readability of Alloy as a drawback with respect to our goals.

In their classical paper, Gligor, Gavrila, and Ferraiolo present a formal specification and composition framework focusing on a variety of separation of duty constraints [25]. They state and prove properties on the relationship of these constraints and on the way they can be composed. The types of separation of duty constraints identified in this work can be used as a starting point for an appropriate set of authorization modules. No specification language is used to defined the framework so that, for example, states need to be handled manually (cf. Section 2.3).

One of the few approaches concentrating on proof-based formal verification of access control models is by Drouineaud, Sohr, et al. [21, 47]. They use the Isabelle theorem prover and a specification in a first-order linear temporal logic. Their main advantage compared to our approach is the application of semi-automatic theorem proving. However, their specifications are lengthy and very difficult to understand. In particular, they do not benefit from having an object-oriented specification language.

The RBAC96 family of models [43], where the ANSI RBAC standard [5] is based on, integrates roles, role-hierarchy, and two variants of separation of duty. Furthermore, the secure role-based workflow models framework [33] merges the RBAC96 concepts with with workflow-related constraints including order of events, execution cardinality restriction, and workflow-based separation of duty. The ANSI standard provides the notion of "functional specification packages" that describe informally how the RBAC components may be combined with each other. From a conceptual point of view these approaches are well thought-out. However, as argued in Section 2, in our view their main drawback is that

[3] The distinction between model-based and proof-based verification is defined by Huth and Ryan [28, p.172].

they are not based on a specification method suitable to handle the complexity and to provide the extensibility necessary for an integration of multiple access control concepts. Further integration of concepts based on the method used will be cumbersome and error-prone.

A further important advantage of our approach to many RBAC models and extensions [43, 37, 33, 25] is that it clearly separates static policy data (which are supposed to be stored in a policy) from the dynamic data such as session data or role activations. Thereby the definition of a policy language is directly supported.

Finally, compared to other high-level access control modeling approaches [32, 9, 38, 43, 33, 11], our authorization modules are one step closer to implementation because they are specified with a dedicated software specification method (cf. Sect. 2.3). For example, the modeling of state transitions can be cumbersome without such a method. Therefore our approach bridges the gap between high-level access control modeling and authorization engine software design.

8 Future Work

Basically, there are three directions for future work: modeling, validation, and implementation.

Firstly, concerning the modeling of access control principles, we strive for the definition of further authorization modules in order to include, for example, obligation, delegation of rights, and further workflow concepts. An additional goal is is to elaborate the innovative aspects we have encountered during the specification of the framework. Also, it may be interesting to see how administrative models can be adapted by the framework. Furthermore, an additional concept that is somewhat orthogonal to all the other concepts mentioned in this report is heterogeneous multi-domain access control. This concept cannot be realized by just adding a further authorization module because more than one authorization engine is involved (e.g. one engine per domain). Instead, a model needs to be built that handles multiple authorization engines which are instances of possibly different access control models. From a high-level perspective, Figure 11 shows how we can realize this in our framework: all participating instances of access control models (i.e. authorization engines at modeling view) need to be aggregated by one multi-domain model specifying how these instances cooperate. In addition, a multi-domain authorization module needs to be introduced to each access control model in order to specify the cooperation internally (such as the handling of external requests).

Secondly, concerning the validation, we strive for an analysis of how we can benefit from existing solutions to automated theorem proving (e.g. based on Isabelle [4]).

Finally, concerning the implementation of an authorization engine according to our generated access control models, we strive for the examination of model-driven development approaches, Object-Z translation techniques and tools [40, 50, 39], as well as approaches based on the Object-Z refinement calculus [52, 20].

Thereby we would strengthen the integration of our framework in the software development process.

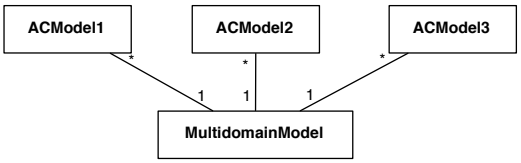

Fig. 11. Multi-Domain Extension (UML)

9 Summary and Conclusions

In this report we presented a framework that builds a foundation for the specification of various RBAC extensions and their integration with each other. In particular, we have already included a role hierarchy, separation of duty, context constraints, and several workflow-related constraints such as history-based separation of duty and prerequisite steps. We have also shown how to deliver policy data models accordingly. Furthermore, it is explained why formal, object-oriented specification in Object-Z is the right vehicle to make the framework extensible, flexible, precise, and capable of handling complexity. We have also demonstrated how these properties are realized in the framework. Finally, we have shown how the specification can benefit from a formal analysis.

As a consequence, we believe that the work of this report can serve as a basis for a unified RBAC reference model bringing the concepts from existing RBAC extensions together. In addition, the framework can be used to articulate new concepts concisely and rapidly without the need of textual descriptions and to analyze these concepts formally. This helps to avoid a faulty design from the beginning on. Finally, since the models of our framework are expressed with a dedicated software specification language, they are highly suitable as a software model for an authorization engine, its communication interface, and its relationships with other components such as a context provider or a workflow management system.

References

1. The ORKA Project Homepage http://www.orka-projekt.de/index-en.htm (2008-04-10).
2. The Alloy Analyzer Project Homepage http://alloy.mit.edu/ (2008-04-10).
3. Object Z Homepage http://www.itee.uq.edu.au/ smith/objectz.html (2008-04-10).

4. The Isabelle/HOL-Z Project Homepage http://www.brucker.ch/projects/hol-z/ (2008-04-10).
5. American National Standard: Role Based Access Control, 2004. ANSI INCITS 359-2004.
6. G.-J. Ahn and H. Hu. Towards Realizing a Formal RBAC Model in Real Systems. In *SACMAT*, pages 215–224. ACM, 2007.
7. C. Alm. An Extensible Role-Based Access Control Model Supporting Advanced Authorization Constraints (Formal Specification), 2008. http://www.informatik.uni-hamburg.de/SVS/ personnel/christopher/pub/calm08rbacmodel.pdf (2008-04-09).
8. A. Anderson. *Core and hierarchical role based access control (RBAC) profile of XACML v2.0*, 2005.
9. J. Bacon, K. Moody, and W. Yao. A model of OASIS Role-Based Access Control and its Support for Active Security. *ACM Trans. Inf. Syst. Secur.*, 5(4):492–540, 2002.
10. E. Bertino, J. Crampton, and F. Paci. Access Control and Authorization Constraints for WS-BPEL. In *IEEE ICWS*, pages 275–284, 2006.
11. E. Bertino, E. Ferrari, and V. Atluri. The Specification and Enforcement of Authorization Constraints in Workflow Management Systems. *ACM TISSEC*, 2(1):65–104, 1999.
12. R. Bhatti, J. Joshi, E. Bertino, and A. Ghafoor. Access Control in Dynamic XMLBased Web Services with X-RBAC. 2003.
13. G. Booch et al. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley, 3rd edition, 2007.
14. R. Botha. *CoSAWoE - A Model for Context-Sensitive Access Control in Workflow Environments*. PhD thesis, 2002.
15. J. P. Bowen. *Formal Specification and Documentation using Z: A Case Study Approach*. International Thomson Computer Press, 1996.
16. I. Craig. *Formal Models of Operating System Kernels*. Springer, 2007.
17. J. Crampton. XACML and Role-Based Access Control. In *DIMACS Workshop on Secure Web Services and e-Commerce*, 2005.
18. J. Crampton and H. Khambhammettu. Delegation in Role-Based Access Control. In *ESORICS*, pages 174–191, 2006.
19. N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The Ponder Policy Specification Language. *LNCS*, 1995:18–39, 2001.
20. J. Derrick and E. Boiten. *Refinement in Z and Object-Z: Foundations and Advanced Applications*. Springer-Verlag, 2001.
21. M. Drouineaud, M. Bortin, P. Torrini, and K. Sohr. A First Step Towards Formal Verification of Security Policy Properties for RBAC. *QSIC*, pages 60–67, 2004.
22. R. Duke and G. Rose. *Formal Object-Oriented Specification Using Object-Z*. Macmillan Press, 2000.
23. R. Duke, G. Rose, and G. Smith. Object-Z: a Specification Language Advocated for the Description of Standards. Technical Report 94–45, The University of Queensland, 1994.
24. D. Ferraiolo and R. Kuhn. Role-Based Access Control. In *15th NIST-NCSC National Computer Security Conference*, pages 554–563, 1992.
25. V. D. Gligor, S. I. Gavrila, and D. Ferraiolo. On the Formal Definition of Separation-of-Duty Policies and their Composition. In *IEEE Symp. on Sec. and Priv.*, pages 172–185, 1998.

26. M. C. Henson, M. Deutsch, and S. Reeves. Z Logic and Its Applications. In D. Bjørner and M. Henson, editors, *Logics of Specification Languages*, pages 489–596. Springer, 2008.
27. M. Hitchens and V. Varadharajan. Tower: A Language for Role Based Access Control. In *POLICY Workshop*, pages 88–106, 2001.
28. M. Huth and M. Ryan. *Logic in Computer Science. Modelling and Reasoning About Systems*. Cambridge University Press, 2nd edition, 2004.
29. D. Jackson. Alloy: A Lightweight Object Modelling Notation. *ACM Trans. on Software Engineering and Methodology*, 11(2):256–290, 2001.
30. D. Jackson et al. A Micromodularity Mechanism. In *ESEC/FSE-9*, pages 62–73.
31. C. B. Jones. *Systematic Software Development Using VDM*. Prentice Hall, 2nd edition, 1990.
32. J. B. D. Joshi, E. Bertino, U. Latif, and A. Ghafoor. A Generalized Temporal Role-Based Access Control Model. *IEEE Trans. on Knowl. and Data Eng.*, 17(1):4–23, 2005.
33. S. Kandala and R. S. Sandhu. Secure Role-Based Workflow Models. In *IFIP Workshop on Database Security*, pages 45–58, 2002.
34. W. kuang Huang and V. Atluri. SecureFlow: A Secure Web-Enabled Workflow Management System. In *ACM RBAC Workshop*, pages 83–94, 1999.
35. N. Li, J. Byun, and E. Bertino. A critique of the ANSI Standard on Role Based Access Control. Technical Report TR 2005-29, Purdue University, 2005.
36. T. Moses et al. eXtensible Access Control Markup Language (XACML) Version 2.0. 2005. OASIS Standard.
37. G. Neumann and M. Strembeck. An Integrated Approach to Engineer and Enforce Context Constraints in RBAC Environments. *ACM TISSEC*, 7(3):392–427, 2004.
38. M. Nyanchama and S. Osborn. The Role Graph Model and Conflict of Interest. *ACM Trans. Inf. Syst. Secur.*, 2(1):3–33, 1999.
39. S. Qin and G. He. Linking Object-Z with Spec#. In *ICECCS '07: Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007)*, pages 185–196. IEEE Computer Society, 2007.
40. S. Ramkarthik and C. Zhang. Generating Java Skeletal Code with Design Contracts from Specifications in a Subset of Object Z. In *ICIS-COMSAR '06: Proceedings of the 5th IEEE/ACIS International Conference on Computer and Information Science and 1st IEEE/ACIS International Workshop on Component-Based Software Engineering,Software Architecture and Reuse*, pages 405–411. IEEE Computer Society, 2006.
41. M. Richters and M. Gogolla. OCL: Syntax, Semantics, and Tools. In T. Clark and J. Warmer, editors, *Object Modeling with the OCL*, pages 42–68. Springer, 2002.
42. R. Sandhu, V. Bhamidipati, and Q. Munawer. The ARBAC97 Model for Role-Based Administration of Roles. *ACM TISSEC*, 2(1):105–135, 1999.
43. R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-Based Access Control Models. *IEEE Computer*, 29(2):38–47, 1996.
44. A. Schaad and J. D. Moffett. A Lightweight Approach to Specification and Analysis of Role-Based Access Control Extensions. In *SACMAT '02: Proceedings of the Seventh ACM Symposium on Access Control Models and Technologies*, pages 13–22. ACM, 2002.
45. G. Smith. A Logic for Object-Z. Technical Report 94–48, The University of Queensland, 1994.
46. G. Smith. *The Object-Z Specification Language*. Springer-Verlag, 2000.

47. K. Sohr, M. Drouineaud, G.-J. Ahn, and M. Gogolla. Analysing and Managing Role-Based Access Control Policies. *IEEE Transactions on Knowledge and Data Engineering*, (to appear).
48. I. Sommerville. *Software Engineering.* Addison Wesley, 7th edition, 2004.
49. J. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, 2nd edition, 1992.
50. V. H. Von. An Environment for Mapping Object-Z Specification into Java/JML annotated code. MSc Thesis, Imperial College London.
51. J. Wainer and A. Kumar. A fine-grained, controllable, user-to-user delegation method in RBAC. In *SACMAT: Proceedings of the Tenth ACM Symposium on Access Control Models and Technologies*, pages 59–66, 2005.
52. J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof.* Prentice-Hall, 1996.

A Used Object-Z Notation

$x : T$	declaration of x as type T
$x : \mathrm{seq}\ T$	x is a sequence of elements of type T
$\land$	logical connective "and"
$\lor$	logical connective "or"
$\Rightarrow$	logical connective "implies"
$\neg$	logical unary connective "not"
$\in$	set membership
$\cup$	set union
$\cap$	set intersection
$\subseteq$	subset
$\forall x : T \bullet P$	for all x of type T, P holds
$\exists x : T \bullet P$	exists x of type T such that P holds
$\mathbb{P}$	power set
$\upharpoonright$	visibility list of a class
Δ	indicates the variables changed by an operation
$R(\!\mid S \mid\!)$	relational image of a relation R under a set S
$\mathbb{B}$	boolean type
$\downarrow$	polymorphic type
?	indicates input variable
!	indicates output variable
$'$	the state of a variable after an operation
$\widehat{=}$	operation definition with schema calculus
$\langle\rangle$	defines a sequence
in	checks subsequence relationship
$\rightarrow$	total function
$\nrightarrow$	partial function
$\leftrightarrow$	relation
$\circ$	backward relational composition
R^{-1}	relational inverse
R^*	reflexive, transitive closure
$\times$	cartesian product
$\copyright$	object containment
$\varnothing$	empty set
$\underline{R}$	infix notation for a relation R