Proving Correctness within an Access Control Evaluation Framework

BY

DIEGO MARTINOIA
Laurea di Primo Livello in Engineering of Computing Systems,
Politecnico di Milano, Milan, Italy, September 2011

THESIS

Submitted as partial fulfillment of the requirements
for the degree of M.S. in Computer Science
in the Graduate College of the
University of Illinois at Chicago, 2013

Chicago, Illinois

Defense Committee:

Lenore D. Zuck, Chair and Advisor

Timothy L. Hinrichs

Pier Luca Lanzi, Politecnico di Milano

ACKNOWLEDGEMENTS

I would like to thank my thesis committee –Lenore Zuck, Tim Hinrichs and Pier Luca Lanzi– for their help in this adventure. Lenore and Tim were a constant source of guidance in developing this work while growing myself, and Pier Luca takes care of the program and flies back and forth over the Atlantic to give his students this opportunity.

DM

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

SUMMARY

This thesis presents the proofs developed to demonstrate correctness of a case study within the Access Control Evaluation Framework (ACEF).
ACEF is a theoretical framework developed at the University of Illinois at Chicago, aimed at creating application-sensitive implementations of access control policies, using well-known access control systems, while preserving some desirable properties.
As the formal proof of these properties, including correctness, is usually a tedious process prone to error, a mixed approach was used, which relies on both human high-level abstraction and insight for the outlining of the proof sketches, and then verifies their formal correctness using a formal prover system.

To achieve this, a generic template for ACEF for the Prototype Verification System formal prover was developed. The process of generating the proofs for this specific case study was also a benchmark for the validity of the template, as a first step towards the realization of a more automated approach to ACEF.

In the conclusions, the validity of the approach is analyzed and possible future steps to improve it are outlined.

CHAPTER 1

INTRODUCTION

Access control, the act of determining who is allowed to do what in a system, is one of the basic pillars of computer security. Due to its central role in this important discipline, access control has been analyzed from a formal point of view by many. Most of the previous works focused on quantifying, in absolute or relative terms, the expressive power of access control schemes[1–10], trying to answer the question "is scheme S more powerful than scheme T?" rather than "is scheme S fit for my specific needs?".

Starting from these considerations, a team of researchers at UIC have been investigating the problem with a different perspective in mind: evaluating access control systems (ACS's) not one against the other, but rather by comparing them against the requirements of a specific application. To address this goal, they developed a theoretical framework called ACEF (Access Control Evaluation Framework). ACEF starts by identifying the specific needs of the application in analysis: once all the required functionality are well-defined, they are represented as a *workload*, a self-contained representation of the abstracted state of the application and of how this state changes during time. One can say that a workload represents the idealized ACS for the application, together with all the non authorization-related information. Then a similar process is carried out for the ACS against which we are testing our application. Once we have both our workload and ACS expressed as finite state automata (FSAs), ACEF lets the user specify the *implementation* of the workload in the given ACS, i.e. the pairing elements that describe how workload states are represented by ACS states, how workload actions are executed using ACS actions and how workload queries are answered using ACS queries. Finally, ACEF formally specifies a (non-comprehensive) list of desirable properties that our implementation should satisfy, so that the user can verify via theorem-proving if these properties hold.

Unfortunately, the formal verification of these properties is far from a trivial challenge: given that ACEF can theoretically represent the pairing of any kind of workload with any kind of ACS, the implementations can become complicated and hard to work with. Therefore, relying on human hindsight and intuition in the proofs can lead to subtle mistakes caused by the overlooking of details. The goal of this work is to show how theorem provers can be used to represent the ACEF framework, with the aim of enforcing a much more rigorous procedure during the development of the proofs, and easing the burden of the analyst who can now sketch a “naive” proof outline by himself, and be assisted by the system during the chewing of all the details, ensuring that nothing is left unchecked.

To achieve this, I developed a generic structure to study ACEF case studies within the Prototype Verification System (PVS). PVS was chosen for its capability to work easily with higher order logic, but as we’ll see the concepts can easily be translated into other systems. Within PVS I also formalized the intuitive notion of correctness by giving it a precise mathematical definition, and went through the proof of correctness for a formalized version of one of the well-known Access Matrix ACS's and the so-called "dynamic coalitions" scenario, described in the next chapter, which was one of the driving needs for the development of ACEF.

As we'll see, the component of correctness related to how workload and ACS states relate to one another is easy to prove in the system, while the part related to how workload actions translate in ACS actions is harder to work with. Typically, this last part of the proof requires to work by induction, as one workload actions translates into a list of ACS actions, and inductions can be complicated as we usually induce over the state elements, and therefore need a different sub-proof for each state element, as we could be "expanding" each one of them.

Anticipating some of the results, I can state that using PVS, or another generic theorem prover, to prove ACEF correctness can be challenging: as every implementation requires an inductive proof whose hypothesis depend on the contents of the workload, completely automating this process would probably require importing work from the field of inductive theorem proving to automatically generate and refine inductive hypothesis. Nevertheless, being constrained by the system allows to avoid ambiguity and subtle mistakes that come natural if one was to prove things in a less formal way. This is, indeed, the burden of being right.

In the remainder of the thesis I'll first outline in more detail the ACEF theory, then move on by introducing the specific application we'll use as case study. After that, I'll describe the choices taken in the development of the PVS structure and their motivation, and finally moving on to the actual proofs. As a final wrap-up, I'll try to outline the lessons learned from the development of this work, and point out the future works direction.

CHAPTER 2

ACEF THEORY

ACEF is a formal, mathematical framework aimed at giving analysts a tool for evaluating ACS's in an *application-sensitive* fashion. In this section I describe a subset of the ACEF theory in order to give the reader the needed background to understand the thesis work. For a complete description, refer to the full paper presenting ACEF[11].

In ACEF we represent both the idealized version of our application (the workload) and the ACS in use as FSA's, specify the pairing elements between them and then verify if this *implementation* satisfies certain properties. The workload and the ACS representations must be self-contained, i.e. they must represent all the required information within themselves, without relying on any outside element.

Consider an application that has a list of its users. The users are real-world entities, peoples or accounts, but they will most likely be identified by a single ID, which in the most general case will be a string over a language ("$user_1$", "$user_2$", etc.). Therefore, the representation of our state becomes a tuple containing a number of sets, either over the basic type *string* or over other sets, and a number of relations among elements of sets or, possibly, other relations. This consideration strongly hints at the possibility to use higher order logic theory, as well as related automated tools, to address ACEF.

Over the representation of a state, we have methods that expose information about the state to the outside. We are going to refer to these methods as *queries*. Queries and states are elements of any generic FSA. Since we are representing systems related to access control, we must add some specificity to the representation. More specifically, the list of queries of our FSA's must include a list of special queries that are related to authorizations. As it's customary in this field, we are going to consider authorization queries in the typical subject-object-right fashion. Below are given some definition that will be useful in explaining the rest of the theory.

2.1 Access Control Model

Definition 1 (Access Control Model)**.**

An Access Control Model (ACM) M is a tuple $\langle S, Q, \models \rangle$ such that:

- S: a finite set of states
- Q: a finite set of queries, including the authorization ones. Queries can have arguments
- R: a subset of Q that specifies the authorization queries
- $\models$: a subset of $S \times Q$ (the entailment relation)

The $\models$ relation specifies which queries are true in which states. There are some important things to note here: first, the queries will always lead to boolean answers, that is to say I can't ask "which user(s) borrowed book X?", but I'll have to ask, for every user, "did user Y borrow book X?". Secondly, queries don't store any semantics by itself, but their behavior is entirely specified by the $\models$ relation. Lastly, note that it's only the presence of the R set that differentiate an ACM from a general-purpose model. We also define accessors to the fields of an ACM $M = \langle S, Q, \models \rangle$ as follows: $States(M)$ denotes S, $Queries(M)$ denotes Q, $Th(s)$ denotes the set of all the queries $q \in Q$ true for a given $s \in S$ (i.e. $s \models q$), and $Auth(s)$ is the set of all the authorization queries $r \in R$ such that $s \models r$ (note that $Auth(s) \subseteq Th(s)$, as $R \subseteq Q$).

Example 1 (Access Matrix type A Model)**.**

Access Matrix type A (AM_a) is the simplest type of access control scheme in the Access Matrix (AM) family. It only consists of a matrix recording triplets that directly represent the authorization policies. So, if we let U denote the set of all possible strings, one possible representation in ACEF of the ACM for AM_a would be:

- States: The states are composed only from the access matrix, which is a subset of U^3. Each $s \in States(A)$ is a possible state of the access matrix. For brevity, I'll refer to the access matrix of state s using $s.m$

- Queries: The only available query is whether or not a triplet belongs to the matrix. Note that the presence of a triplet in the matrix also means that the triplet represent an authorization record, therefore this is one of those cases where $R = Q$
- Entailment: $s \models q \iff q \in s.m$

An ACM is a specification of how to store and query information, but it doesn't allow to ways to change the current state in the desired one. An ACS is an extension of an ACM that also allow the possibility to evolve the states. From a mathematical perspective, an ACS transforms the set of states of an ACM into a labelled directed graph, adding labelled edges that represent how actions affect the states.

2.2 Access Control System

Definition 2 (Access Control System)**.**

An ACS Y is a tuple $\langle M, L, next \rangle$ such that:

- M: an ACM, as defined in the previous paragraph.
- L: a finite set of labels (also called commands or actions). Labels can have arguments
- $NEXT$: a total function $States(M) \times L \rightarrow States(M)$ (the transition function).

Also in this case, we have some accessors: $Labels(Y)$ is L, $Queries(Y)$ is $Queries(M)$ and $States(Y)$ is $States(M)$. Also, for a finite sequence of labels $l_1 \bullet ... \bullet l_n$ we define $TERMINAL(s, l_1 \bullet ... \bullet l_n)$ as the state reached by applying one by one the labels starting from state s.

Example 2 (Access Matrix type A System)**.**

The only available commands in AM_a are related to adding or removing triplets to or from the authorization matrix. Therefore:

- $addMatrix(\langle s, o, r \rangle)$: adds the triplet $\langle s, o, r \rangle$ to the access matrix.
- $removeMatrix(\langle s, o, r \rangle)$: removes the triplet $\langle s, o, r \rangle$ to the access matrix.

The $NEXT$ function has to take care of the subtle details. For $addMatrix$, the reached state is different from the starting one only if the added triplet wasn't present already. For $removeMatrix$, the reached state is different only if the removed triplet was present in the matrix. Remember that $NEXT$ is a total function, and therefore there aren't "illegal" actions, just commands that will not modify the state in certain situations.

2.3 Workload

Now that ACSs are defined, let's switch our attention to defining the workloads.
As stated earlier, a workload is an idealized, self-contained representation of the application against which we want to test the ACS. In the original formulation of ACEF, a workload differs from an ACS by adding a set of traces, i.e. the subset of sequences of commands, starting from a given state, that are relevant to our needs. This is done to reduce the amount of work required to prove the desired properties. Since we aim at having the machine deal with most of the proofs, we do not have to restrict the traces and can allow for similar definitions of ACSs and workloads. This similarity is very important to facilitate the

analysis, however, in spite of the similarity in their structures, ACSs and workloads have a very different meaning behind them: the workloads represent our idealized application, something that we have envisioned as designers. The ACS is thought to be an off-the-shelf system, something that we have to use as-is to meet the needs of our application.

For a running example, this work uses the case study of “dynamic coalitions”. This case study was chosen due to a pair of report[12,13] that demonstrates how the U.S.A. system of storing and managing sensitive information is inappropriate for those cases, always more frequents, where coalitions of organizations, among which sensitive information must be shared, are formed and disbanded often. In our terminology, a state of the workload is given by:

- *Subjects*: a finite set of subjects ID (Alice, Bob, etc.)
- *Objects*: a finite set of objects ID ($document_1$, $document_2$, etc.)
- *Rights*: a finite set of rights ID (read, write, etc.)
- *Organizations*: a finite set of organizations ID (France, U.N., etc.)
- *Auth*: a finite set of $\langle s, o, r\rangle$ triplets (where $s \in \mathit{Subjects}, o \in \mathit{Objects}, r \in \mathit{Rights}$) that specify the current authorization policy.
- *Belongs*: a function $\mathit{Subjects} \rightarrow (\mathit{Organizations} \cup \perp)$ that specifies the affiliation of subjects to the different organizations. The definition of the function *Belongs* implies

that at any time subjects are either unaffiliated (hence the $\perp$ symbol) or affiliated with at most one organization.

The commands available in our case study are just two, which are given here together with the informal description of their $NEXT$ function. We'll later see how this informality leads to non-trivial issues during the proving steps.

- $joinCoalition(\mathit{org}, \mathit{newAuth})$: for each $\langle a, b, c\rangle \in \mathit{newAuth}$, (i) add $Auth(a, b, c)$ and (ii) change $Belongs$ such that $Belongs(s) = \mathit{org}$
- $leaveCoalition(\mathit{org})$: for each subject s such that $Belongs(s) = \mathit{org}$, (i) revoke all the authorizations of s and (ii) change $Belongs$ such that $Belongs(s) = \perp$

Note that we are assuming that *newAuth* is a finite set of triplets, while *org* is a single Organization ID. What we want to achieve is that the $joinCoalition$ commands allow to specify the entering of an organization into a coalition, together with their desired authorization policy, while the $leaveCoalition$ commands remove an organization from the coalition and void all the rights of its members.

2.4 Implementation

The final piece of ACEF is given by the *implementation*. An implementation is a specification of three specific functions that define how workload and ACS are linked together.

Definition 3 (Implementation)**.**

For a workload W and an ACS Y, an implementation is a triplet of functions $\langle \sigma, \alpha, \pi \rangle$:

- $\sigma : \mathit{States}(W) \rightarrow \mathit{States}(Y)$ (state-mapping)
- $\alpha : \mathit{States}(Y) \times \mathit{Labels}(W) \rightarrow \mathit{Labels}(Y)^*$ (action-mapping)
- $\pi : \mathit{Queries}(W) \times \mathit{powerset}(\mathit{Queries}(Y)) \rightarrow \mathit{booleans}$ (query-mapping)

σ, the state-mapping, specifies how we can construct, starting from a workload state, the ACS state that represent it. α, the action-mapping, specifies how a workload action is translated in a finite sequence of ACS actions. Finally π, the query-mapping, specifies how a workload query in a given workload state can be answered as true or false given the theory of the mapped ACS state, i.e. the set of all queries true in that state. Note that, in the general formulation of π the states do not appear. This is because, in ACEF, π is only used for the formulation of the correctness property, which we'll see in a bit, which already includes the states information.

2.5 Correctness

ACEF comes together with a set of desirable properties already mathematically defined. The most important of them is by far *correctness*. While the other properties may guarantee case-specific interesting behaviours, correctness is the *conditio sine qua non* an implementation is simply wrong. Correctness holds when the implementation elements "play well" along with each other, i.e. state, action and query mapping are not inconsistent.

Definition 4 (Correctness)**.**

For a given ACEF case instance composed by $\langle W, Y, \sigma, \alpha, \pi\rangle$ where W is a Workload, Y is an ACS, and σ, α, π an implementation, correctness holds iff:

(Query-mapping preservation)

$$\forall (s \in States(W), \quad q \in Queries(W)) :$$

$$q \in Th(s) \iff \pi(q, Th(\sigma(s))$$

$$\wedge$$

(Action-mapping preservation)

$$\forall (s \in States(W), l \in Labels(W) :$$

$$\sigma(NEXT(s,l)) = TERMINAL(\sigma(s), \alpha(\sigma(s), l))$$

Correctness is the logical conjunction of two other properties. The first property, called "*query-mapping preservation*", guarantees that a workload query is true in a workload state only if this truth can be derived just as well by looking at the theory of the mapped ACS state. From a real-world point of view, this means that you can always derive the truths of your idealized application by looking at the state of your data structures. The second property, "*action-mapping preservation*", is related to the pairing of states and actions of the workload and the ACS. Formally, this properties states that applying a workload action l starting from a workload state s will lead you in a workload state $NEXT(s,l)$ which, when mapped to the ACS, will be the same you would reach by starting from the state to which s is mapped and applying the actions chain that represent the translation of

l in the ACS labels. Informally, the properties states that the actions paths in the workload and in the ACS are always consistent.

This exposition covers all the details of ACEF needed to understand this work. As already stated, ACEF actually offers a wider number of formalized properties, as well as reduction guarantees among implementations, but they go beyond the scope of this work. If the reader is interested in further details, he can consult the original full paper on ACEF[11].

CHAPTER 3

PVS STRUCTURE

In this chapter I'm going to outline the implementation of the ACEF theory in the Prototype Verification System (PVS). First, I'll give a brief overview of how PVS works, then I'll go through the details of the code, pointing out where necessary the inner representation mechanics of the system and the rationale behind the design decisions taken. This chapter was written as a guide to understanding the code in the appendix. Albeit snippets are provided wherever this is possible, it is therefore suggested to read the chapter with a copy of the code at hand.

3.1 PVS Overview

PVS is a "verification system", i.e. an integrated solution that comprehends a specification language and a theorem prover. It was developed by SRI International[1] and is now freely available on the web[2]. PVS comprehends a specification language, a number of predefined theories and proved theorems (the "*prelude*"), a type checker and an interactive prover, as well as some useful add-ons like "pretty-printing" of the code into portable document format (.pdf) files. PVS was built summing up 25 years of experience in the

[1] http://www.sri.com/

[2] http://pvs.csl.sri.com/

field of specification and property verification, and is nowadays used in many sectors of application, whenever it's critical to prove system properties. For example, NASA itself organizes yearly summer schools on their use of PVS and their huge custom library of theories. PVS works by letting the user define a formal specification of the system in exam, offering native support for various common data structures such as records, lists and sets, as well as common mathematical concepts like quantifiers, types algebra and the like. PVS operates in a higher order logic regime, where functions can take as argument and return other functions. After the description of the system is specified, PVS lets the user specify a number of lemmas that need to be proven, given the specification. Whatever the form of the lemma, PVS converts it in and equivalent form:

$$A_1 \wedge A_2 \wedge ... \wedge A_n$$

$$\Rightarrow$$

$$B_1 \vee B_2 \vee ... \vee B_m$$

Where the A_i atoms are called "antecedents" and the B_j atoms are called "succedents". Within the PVS prover, the user can then expand and manipulate definitions and formulas to reach a trivial truth, i.e. where one of the following conditions is met:

- The conjunction of the antecedents is trivially false
- The disjunction of the succedents is trivially true
- One of the succedents is the same as one of the antecedents

All of PVS transformation guarantee to preserve the correctness of the process, and they include some fairly powerful tools for the most common cases. For example, proving something via induction over the naturals is rather trivial and automated, while user effort is required for the most "exotic" cases. As we'll see, *dynamic coalitions* is one of the latter kind.

3.2 Issues

The main issue found in using PVS for ACEF was related to the data structures. As I mentioned already, PVS offers a number of default data structure. For the specific nature of ACEF, the ones of interest are the finite versions of sets, lists and sequences. As we'll see, each of these has some some, but not all, of the properties relevant to ACEF, and therefore the development of custom data structures was required, by taking something from each of the native options.

In PVS, sets are equivalent to their characteristic function, i.e. a set of elements of type T is nothing but a function $f : T \rightarrow booleans$ or, in PVS notation, $f : [T \rightarrow booleans]$. Sets have the interesting property of uniqueness, i.e. it's impossible to store duplicates within a set, This is something desirable to have for ACEF, as most of our collections represent ID collections, where duplicates are not allowed. Unfortunately, sets lack ordered access, i.e. there is nothing like a "first", "second" or "last" member of a set[1]. Since ACEF works

[1]Actually, the $choose$ command of PVS offers something similar to an ordering over a set, but it's not fully appropriate for our needs.

closely related to the path traversal of a graph, the ordering is of extreme importance.

Lists are right-recursive data structures, in the sense that a list of elements of type T can be composed by pre-pending an element of type T to an existing list of elements of type T, with the base case of an empty list. Lists offer a total ordering among their elements, and their recursive structure matches perfectly the functional paradigm of the PVS programming style. Unfortunately, lists can contain duplicates, and it's impossible to access a definite position without traversing the list.

Finally, sequences of type T are defined as records containing two fields: their current length l, which is a natural number, and a function from the subset of naturals below l to elements of type T, which is used for positional access. Sequences too define a total ordering among their elements, and allow positional access as well as knowledge of their length without the need of traversing them. But they can store duplicates, and being records they can't be used recursively in a natural way.

Note that, to some, uniqueness may seem like an unnatural property to desire: it's easy to imagine that lists of commands may contain duplicates. Albeit this is true only for the commands lists, as sets naturally offer uniqueness, I enforced it also on the collections with in mind the possibility of expanding, in the future, this work towards other ACEF properties. Among these, we find "*safety*", which informally states that intermediate states during the execution of sequences of commands do not remove/assign rights more times than those strictly needed (for example, to remove a right, we have to remove it only once and not issue sequences like remove/add/remove). Albeit it's still possible to break safety with a list of unique commands in certain special cases, the impossibility of having duplicate commands in a single sequence is a strong deterrent for the analyst to try and infringe safety. After all, it's still possible to realize correct implementations that do not need duplicate commands in a single sequence. If it was the case where this was particularly needed, the analyst might just specify the data structures of this unlikely case as a native sequence or list.

Since all the native possibilities were not ideal, I decided to develop my custom data structures that would possess all the desirable properties required to make proofs easier.

3.3 ACEF Data Structures

First of all, let's recap which properties our ideal structures might need:

- Uniqueness: the collections must not allow duplicate elements
- Ordering: a precise iteration order of the elements must be defined
- Composability: the operators of the collections must preserve both the ordering and the uniqueness of the elements
- Flexibility: it is desirable that the collection allows for both recursive and positional access and composition

Given these premises, I customized the native sequence structure, and its operators, in such a way to preserve uniqueness of the contained elements and recursive composition

An ACEF data structure of elements of type T is a tuple $\langle length, seq \rangle$ where:

$$\begin{aligned}
&\mathit{length} \in \mathbb{N},\\
&\mathit{seq} : (\mathbb{N} \leq \mathit{length}) \rightarrow T \quad | \\
&\quad \forall (n_1, n_2 \in (\mathbb{N} \leq \mathit{length})) : \\
&\quad n_1 \neq n_2 \iff \mathit{seq}(n_1) \neq \mathit{seq}(n_2)
\end{aligned}$$

```
t_Seq: TYPE =
[#
        length: nat,
        seq: {s: [below[length]->T] |
        (FORALL (n1: below[length], n2: below[length]):
        n1 /= n2 IFF s(n1) /= s(n2))}
#]
```

Figure 1: ACEF data structure

and traversal. As reported in the appendixes, the operators were modified in order to only accept collections and elements that respect constraints that guarantee the properties. For example, the concatenation of two ACEF sequences impose that no element is shared between the two collections, to guarantee uniqueness in the result.

The specification of the ACEF custom structure, without including all the specific properties and operators, is given in figure 1. Note how uniqueness is intrinsic in the definition. The development of these structures wasn't the result of a waterfall-like process, but rather the outcome of an iterative design refinement, where I had to go through all the proofs from top to bottom multiple times, and see what I needed next to get to the final demonstrations: PVS doesn't make discounts, if something isn't specified somewhere, you can't just take it for granted as we'd do using our human hindsight.

Since ACEF holds the possibility to analyse basically any kind of workload-ACS implementation, it's possible that future cases will require additional properties, lemmas or operators, and therefore the specification of the data structures, as well as that of the whole ACEF implementation, is not to be considered completed, but always expandable.

3.4 ACEF Implementation

In this section I dive in the implementation of ACEF in PVS. When non-intuitive, I'll point out the details in the syntax of the PVS language. For a much more comprehensive reading on the matter, the reader can consult the PVS vast documentation[1].

ACEF assumes that the ACS and workload alphabets are the same, and therefore exists a set $\mathbb{U}$ of all possible strings in common between the workload and the ACS. In the specific case in analysis, I partitioned the set $\mathbb{U}$ into two disjoint sub-sets: the "usable" strings, i.e. the set of all strings that can be used from the user to define his own ID's, instances, etc., and the set of "keywords", i.e. reserved words that cannot appear in the data. In our case, the keywords set is actually just a singleton composed of the so-called *"skolem"* constant. Albeit specifying a set of reserved words is very likely to break some

[1]http://pvs.csl.sri.com/documentation.shtml

Let $\mathbb{U}$ the set of all strings.
Define over $\mathbb{U}$ two predicates, t_Strings_Pred and t_Skolems_Pred.
Disjointedness is achieved by adding the axiom:
$\forall(s \in \mathbb{U})$:
$t\_Strings\_Pred(s) \iff \neg t\_Skolems\_Pred(s)$

```
t_ALL_Strings: TYPE+

t_Strings_pred: [t_ALL_Strings -> bool]
t_Strings: TYPE+ = (t_Strings_pred)

t_Skolems_pred: [t_ALL_Strings -> bool]
t_Skolems: TYPE+ = (t_Skolems_pred)

Strings_Nature_Axiom: AXIOM
(
        FORALL (s: t_ALL_Strings): t_Strings_pred(s) IFF NOT t_Skolems_pred(s)
)
```

Figure 2: Hierarchy of string types

of the other ACEF properties, first of all the “homomorphism” property, our case-study only cares about correctness and therefore our implementation using keywords is a valid option. Disjointedness was achieved by specifying an axiom, a lemma that does not need to be proven, that states that the predicates that characterize the two sub-types are one the opposite of the other. This hierarchy can be found in figure 2.

3.4.1 Workload

We have seen in the previous chapter that, in general, to define a workload the specification of five main elements is required: states, queries, labels, entailment relation together with the theory of the states and, finally, the transition function. In this subsection

```
% DEFINITION
t_Record_Type =
[#
        field_1: t_Field_1,
        field_2: t_Field_2,
        field_3: t_Field_3
#]

% INSTANTIATION
my_record =
(#
        field_1:= value_1,
        field_2:= value_2,
        field_3:= value_3
#)

% ACCESS. THIS IS TRUE:
my_record`field_1 = value_1
```

Figure 3: Records syntax

I'm going to show, for each of these elements, how they can be represented in general, and then refer to the specific implementation of the case in analysis.

3.4.1.1 Workload States

A state is nothing but a collection of other elements, usually sets or relations. Therefore, the most appropriate data structure appears to be a record. Note that a tuple, i.e. a record with positional access rather than named access, would work too, but the readability of the code would deeply be negatively affected. The PVS syntax for records is a bit peculiar. An example of record type definition, record instantiation and access is given in figure 3. In general, as the workload should have its own authorization policy, we can expect the presence of at least the subjects, objects and rights sets, as well as some form of representation of the authorization policy.

In the previous chapter, we have seen that the dynamic coalitions workload stores two main elements: the authorization policy, composed of $\langle subject, object, right\rangle$ triplets, and the affiliation function, that specifies for each user at most one organization to which he's affiliated. In addition to that, the workload also specifies a number of finite sets of subjects, objects, rights and organizations, which in our specific case are immutable, also because the addition of the possibility for these sets to change with time only increase the amount of proofs to be done by adding trivial lemmas that do not bring any hindsight with them. For the affiliation function, we need to specify that a "special" organization exists that represent the non-affiliation of a subject. Note that this is just one of the many ways to achieve the representation of the non-affiliation: one could have decided to create a sub-set of subjects containing only the affiliated subjects and defining the domain of the affiliation function over that set rather than over all the subjects. Workload states store subjects, objects, rights and organizations as disjoint finite sets over the strings. The authorization policy is a finite set of subject-object-right triplets, while the affiliation (belongs) function is a function from subjects to organizations. The organizations set contains the special *unaffiliated* element. There are also some constraints that help during the proofs, such as the disjointedness of subjects, objects, rights and organizations, and the fact that triplets are S-O-R in the authorization policy and that affiliation pairs are indeed S-Org. The code for the workload states is given in figure 4.

3.4.1.2 Workload Queries

In general, queries are elements that have arguments. They usually are of more than onr type, and each type usually has a different signature. Given these characteristics, the data structure that better seems to match them is that of a "*DATATYPE*", i.e. a data structure with different types of constructors, that represents the disjoint union of its constructors instances. In this way, we can define the super-set of all queries as the disjoint union of different sub-types, each one with its own arguments, and be able for any query to know its sub-type and access its arguments. Again, we expect in the general case to have at least the authorization-related queries.

In our case, we only have two possible queries: "is this triplet part of the authorization policy?" and "is this subject-organization pair part of my affiliation function?". Remember that queries should, in ACEF, return only boolean answers, and therefore the affiliation queries cannot be of the form "to which organization is this subject affiliated?". In general, we might want to add also queries for the subjects, objects, rights and organizations sets, but they are of scarce interest, especially in PVS where the sets coincide with their characteristic functions. The specification of the workload queries is given in figure 5.

Let $\mathbb{S}$ the set of all usable strings.
Given two sets A and B, let $disjoint(A, B) \iff A \cap B = \emptyset$
A workload state is a tuple $\langle subjects, objects, rights, auth, organizations, belongs\rangle$ where:

subjects $\subset \mathbb{S}$,

objects $\subset \mathbb{S} \quad | \quad disjoint(objects, subjects)$

rights $\subset \mathbb{S} \quad | \quad disjoint(rights, objects) \wedge disjoint(rights, subjects)$,

auth is a finite set of $\langle e_1, e_2, e_3\rangle$ triplets $\subset \mathbb{S}^3 \quad | \quad e_1 \in subjects \wedge e_2 \in objects \wedge e_3 \in rights$,

organizations $\subset \mathbb{S} \quad |$

$disjoint(organizations, rights) \wedge disjoint(organizations, objects) \quad \wedge$

$disjoint(organizations, subjects) \wedge unaffiliated \in organizations$,

belongs : $subjects \rightarrow organizations$

```
t_States_w : TYPE =
[#
        subjects:       t_Subjects_w,
        objects:        {obj: t_Objects_w | disjoint?(obj, subjects)},
        rights:         {rig: t_Rights_w |
                                disjoint?(rig, objects) AND
                                disjoint?(rig, subjects)},
        auth:           {aut: t_Auth_w | (FORALL (a: t_TripleString):
                                member(a, aut) IMPLIES
                                member(a`e1, subjects) AND
                                member(a`e2, objects) AND
                                member(a`e3, rights))},
        organizations:  {org: t_Organizations_w |
                                        disjoint?(org, rights) AND
                                        disjoint?(org, objects) AND
                                        disjoint?(org, subjects) AND
                                        member(unaffiliated, org)},
        belongs:        {bel: t_Belongs_w | (FORALL (b: t_Strings):
                                IF member(b, subjects) THEN
                                member(bel(b), organizations)
                                ELSE
                                        bel(b) = invalid
                                ENDIF)}
#]
```

Figure 4: States of the workload

Let $\mathbb{S}$ the set of all usable strings.
The type of workload queries is the disjoint union of two sub-types of workload queries.
Authorization queries take as argument a triplet of usable strings $aut = \langle e_1, e_2, e_3 \rangle \quad | \quad e_1, e_2, e_3 \in \mathbb{S}$
Affiliation queries take as argument a couple of usable strings $bel = \langle e_1, e_2 \rangle \quad | \quad e_1, e_2 \in \mathbb{S}$
workload queries = authorization_queries $\dot\cup$ affiliation_queries

```
t_Queries_w  :  DATATYPE
BEGIN
          authCons_w(
                    aut:  t_TripleString
          ):  auth_w?

          belongsCons_w(
                    bel:  t_DoubleString
          ):  belongs_w?
END  t_Queries_w
```

Figure 5: Queries of the workload

3.4.1.3 Workload Labels

Labels are the actions available in the workload. As queries, they can be of different types with different arguments, and are therefore defined using again a *DATATYPE* construct. In the general case, we expect the presence of some mechanism to at least be able to add and remove authorization policy elements, albeit these mechanism may be indirect, as in the case in analysis.

In the dynamic coalitions scenario we only have two possible workload actions: *joinCoalition(org, newAuth)* and *leaveCoalition(org)*. Again, it should be in general possible also to modify all the elements of the state, and therefore labels like *addSubject* or *re-*

Let $\mathbb{S}$ the set of all usable strings.
The type of workload labels is the disjoint union of two sub-types of workload labels.
joinCoalition labels take as argument an usable string *org* $\in \mathbb{S}$ and an ACEF data structure *newAuth* of triplets of usable strings.
leaveCoalition labels take as argument an usable string *org* $\in \mathbb{S}$.
workload labels = joinCoalition_labels $\dot{\cup}$ leaveCoalition_labels

```
t_Labels_w : DATATYPE
BEGIN
        joinCoalitionCons_w(
                org      : t_Strings,
                newAuth  : t_Seq [t_TripleString]
        ): joinCoalition_w?

        leaveCoalitionCons_w(
                org      : t_Strings
        ): leaveCoalition_w?
END t_Labels_w
```

Figure 6: Labels of the workload

moveObject should be available, but they aren't interesting here. Recall from the previous chapter that *joinCoalition(org, newAuth)* represents the joining of the coalition from organization *org* and specifies its authorization policy via the *newAuth* argument. *org* is a simple string element, while *newAuth* is an ACEF collection of triplets of strings. *leaveCoalition(org)* instead removes an organization from the coalition, removing all its affiliations and voiding the rights of its members. Here too, *org* is a string. The specification of the workload labels is given in figure 6.

3.4.1.4 Workload Entailment and Theory

The entailment relation specifies the semantics of the queries. It is implemented as a finite-set of state-query pairs, specified according to the semantics of the queries. The theory operator instead is just an interface for the finite set of queries in entailment with a state.

In this case, an authorization query is true in a state iff its triplet is part of the authorization policy of the state, and an affiliation query is true in a state iff its couple represent a pairing of the affiliation function of the state. The theory of a state is nothing but the finite set of all the queries that are entailed with it. The definition of the entailment and theory of the workload is given in figure 7.

3.4.1.5 Workload Transition Function

The NEXT transition function specifies the semantics of the labels, i.e. how a state is modified by applying a command. For simplicity, the general structure returns the specification of a whole new instance of state, somehow related to the starting one. In other words, states are considered immutable. In our scenario, the *joinCoalition* label modifies the state by adding all the *newAuth* elements to the state's authorization policy, and changing the affiliation function so that all the subjects appearing in a *newAuth* element are now affiliated with the joining organization, leaving the affiliation of the other subjects unchanged. The *leaveCoalition* commands instead removes all the rights of the subjects affiliated with the exiting organization, and then marks those subjects as unaffiliated, leav-

The $\models_w$ relation of a workload w is a finite set of tuples $\langle state, query \rangle$ where:

state $\in States(w)$,

query $\in Queries(w)$,

The theory of a workload state s is the finite set of workload queries $q \mid \langle s, q \rangle \in \models_w$

```
t_Entailments_w : TYPE =
[#
        state : t_States_w,
        query : t_Queries_w
#]

ENTAILMENT_w : finite_set[t_Entailments_w] =
(LAMBDA (e: t_Entailments_w):
        CASES e`query OF
                authCons_w(aut): member(aut, e`state`auth),
                belongsCons_w(bel): e`state`belongs(bel`e1) = bel`e2
        ENDCASES
)

THEORY_w(state_w : t_States_w) : finite_set[t_Queries_w] =
(LAMBDA (query_w : t_Queries_w) :
        ENTAILMENT_w((# state:=state_w, query:=query_w #))
)
```

Figure 7: Entailment of the workload

ing the rights and affiliations of the other subjects unchanged. The related code is not reported for space reasons and can be found in the appendix.

3.4.2 ACS

3.4.2.1 ACS State

Similarly to what happens with the workload state, a record structure is used to represent states. But in this case, for the generic version, we don't expect the authorization policy to be explicitly stored in the state, but rather it will be derived from the existing information, via queries and the entailment relation.

In our case study, the proposed ACS is a version of Access Matrix type A (AM_a). Recall from the previous chapter the definition of AM_a. The totality of the state is represented just by a matrix, i.e. a finite collection of string triplets. In our case, the ACS state records only contain one field, which is indeed a finite set of triplets. Note that, due to the PVS strong types system, the third element of our triplets can be either of $t\_Strings$ type or of $t\_Skolems$ type. This is because we are going to store both the workload authorization policy and affiliation function in the matrix, and the presence of the $skolem$ keyword will be the element that allows us to distinguish among the two types of information. The specification of ACS states is given in figure 8.

Let $\mathbb{U}$ the set of all strings.

Let $\mathbb{S} \subset \mathbb{U}$ the set of all usable strings.

An AM_a ACS state is a tuple $\langle matrix \rangle$ where:

  matrix is a finite set of $\langle e_1, e_2, e_3 \rangle$ triplets |

    $e_1, e_2 \in \mathbb{S} \wedge e_3 \in \mathbb{U}$

```
t_Matrix_Triplet: TYPE =
[#
        e1: t_Strings,
        e2: t_Strings,
        e3: t_ALL_Strings
#]

t_Matrix_y  : TYPE = finite_set[t_Matrix_Triplet]

t_States_y  : TYPE =
[#
        matrix: t_Matrix_y
#]
```

Figure 8: States of the ACS

Let $\mathbb{U}$ the set of all strings.
Let $\mathbb{S} \subset \mathbb{U}$ the set of all usable strings.
The type of the ACS queries is the disjoint union of two sub-types of ACS queries.
Authorization queries take as argument a triplet of usable strings $aut = \langle e_1, e_2, e_3 \rangle \quad | \quad e_1, e_2, e_3 \in \mathbb{S}$
Matrix queries take as argument a triplet of strings $mat = \langle e_1, e_2, e_3 \rangle \quad | \quad e_1, e_2 \in \mathbb{S} \wedge e_3 \in \mathbb{U}$
ACS queries = authorization_queries $\dot{\cup}$ matrix_queries

```
t_Queries_y : DATATYPE
BEGIN
        authCons_y(
                aut : t_TripleString
        ): auth_y?

        matrixCons_y(
                mat : t_Matrix_Triplet
        ): matrix_y?
END t_Queries_y
```

Figure 9: Queries of the ACS

3.4.2.2 ACS Queries

The queries of the ACS relates to the specific case. In general, we expect to have an authorization-related query, but the specific ones depend on the ACS in use.

In our scenario, the available ACS queries are those related to the ACS authorization policy, as well as those related to the ACS structure. In the special case of AM_a, these two elements collapse in the same one, but in the general framework they are two semantically different things and are therefore differentiated. The definition of the ACS queries is reported in figure 9.

Let $\mathbb{U}$ the set of all strings.
Let $\mathbb{S} \subset \mathbb{U}$ the set of all usable strings.
The type of the ACS labels is the disjoint union of two sub-types of ACS labels.
addMatrix labels take as argument a triplet of strings $mat = \langle e_1, e_2, e_3 \rangle \quad | \quad e_1, e_2 \in \mathbb{S} \wedge e_3 \in \mathbb{U}$
removeMatrix labels take as argument a triplet of strings $mat = \langle e_1, e_2, e_3 \rangle \quad | \quad e_1, e_2 \in \mathbb{S} \wedge e_3 \in \mathbb{U}$
ACS labels = addMatrix_labels $\dot{\cup}$ removeMatrix_labels

```
t_Labels_y : DATATYPE
BEGIN
        addMatrixCons_y(
                mat : t_Matrix_Triplet
        ): addMatrix_y?

        removeMatrixCons_y(
                mat : t_Matrix_Triplet
        ): removeCoalition_y?
END t_Labels_y
```

Figure 10: Labels of the ACS

3.4.2.3 ACS Labels

In the general case, we expect to have all the required commands needed to alter the state, and therefore there are no generic elements, but all of the labels are case-specific. Note that the authorization policy must be derived from the other structures, and therefore there won't be available commands to alter it directly.

In this case, the only two things that we can do in AM_a are adding and removing triplets from the matrix. The specification for the ACS labels can be seen in figure 10.

3.4.2.4 ACS Entailment and Theory

Similarly to what happens in the workload, entailment and theory are defined respectively as a set of state-query pairs and a function from a state to a set of queries.

As mentioned already, in AM_a the difference between authorization-related and matrix-related is almost only syntactic. Authorization queries are true iff the authorization triplet is present in the matrix, and matrix queries are true iff the matrix contains the argument triplet. Nevertheless, note that those matrix triplets that represent affiliation information do not affect the authorization queries, as the third element of affiliation triplets is always the $skolem$ keyword, which type is not of $t\_Strings$ but $t\_Skolems$, and therefore authorization queries cannot be asked regarding affiliation triplets. The code is reported in figure 11.

3.4.2.5 ACS Transition Function

The semantics of the ACS labels is quite intuitive: *addMatrix* adds a triplet to the matrix, *removeMatrix* removes a triplet from the matrix. The code can be found in figure 12.

3.4.2.6 ACS Path-traversing Function

ACEF defines the existence of a TERMINAL function for ACS states, that specifies the state reached starting from a given state and applying a sequence of commands in the given order. $TERMINAL_y$ is just the recursive specification of this concept. Note that, according to the cases, it might be more useful to specify TERMINAL as a left-recursive or

The $\models_y$ relation of an ACS y is a finite set of tuples $\langle state, query\rangle$ where:

state $\in States(y)$,

query $\in Queries(y)$,

The theory of an ACS state s is the finite set of ACS queries $q \mid \ \langle s, q\rangle \in \ \models_y$

```
t_Entailments_y : TYPE = [# state: t_States_y, query: t_Queries_y #]

ENTAILMENT_y : finite_set[t_Entailments_y] =
(LAMBDA (e: t_Entailments_y):
         CASES e`query OF
                 authCons_y(aut):
                         member(aut, e`state`matrix),
                 matrixCons_y(mat):
                         member(mat,  e`state`matrix)
         ENDCASES
)

THEORY_y(sy : t_States_y) : finite_set[t_Queries_y] =
(LAMBDA (qy : t_Queries_y) :
         ENTAILMENT_y((# state:=sy, query:=qy #))
)
```

Figure 11: Entailment and theory of the ACS

The transition function of an ACS y is a function
$NEXT : States(y) \times Labels(y) \to States(y)$

```
NEXT_y (state: t_States_y, label: t_Labels_y) : t_States_y =
CASES label OF
        addMatrixCons_y(mat):
                (#
                        matrix := add(mat, state`matrix)
                #),
        removeMatrixCons_y(mat):
                (#
                        matrix := remove(mat, state`matrix)
                #)
ENDCASES
```

Figure 12: Transition function of the ACS

right-recursive function, but it's hard to foresee which is the best way before dealing with the actual proofs. Also note that the path-traversing function is always the same, with the exception of being left or right recursive, and doesn't change w.r.t. the ACS in use. The code is reported in figure 13.

3.4.3 Implementation

As mentioned already, an implementation for ACEF consists in the specification of the three α, σ, π functions that specify how the ACS and the workload are linked together. More specifically, α specifies how a workload action is translated in a sequence of ACS commands, σ specifies how a workload state is translated in an ACS state and π defines how workload queries can be answered looking at the truths of the ACS.

The path-traversing function of an ACS y is a function

$TERMINAL : States(y) \times Labels(y)^{*} \rightarrow States(y) \quad |$

$\quad TERMINAL(s, empty\_seq) = s \quad \wedge$

$\quad TERMINAL(s, seq \bullet l) = NEXT(TERMINAL(s, seq), l)$

```
TERMINAL_y (sy: t_States_y, ly_star: t_Seq[t_Labels_y]):
RECURSIVE t_States_y =
        IF ly_star'length = 0 THEN
                sy
        ELSE
                NEXT_y(TERMINAL_y(sy, allButLast(ly_star)),last(ly_star))
        ENDIF
MEASURE ly_star'length
```

Figure 13: Path-traversing function of the ACS

3.4.3.1 State-Mapping

Let's start from σ. The state mapping for our case study is limited to the translation of the authorization policy and of the affiliation function. Actually, one should define how subjects, objects, rights and organizations are encoded in the ACS too, but these mappings are not of particular interest in our case study and were skipped for simplicity. The state mapping simply states that the authorization policy triplets are put "as-is" within the matrix, while the affiliation couples $belongs(sub) = org$ are reported as $\langle sub, org, skolem \rangle$ for all the affiliated subjects. Non-affiliated subjects do not have an ACS affiliation tuple. Recall that in our case the matrix is a finite set, and sets in PVS are the same as their characteristic function, i.e. a predicate over their type. The *LAMBDA* notation used is the

Let $\mathbb{U}$ the set of all strings.
Let $\mathbb{S} \subset \mathbb{U}$ the set of all usable strings.
The state-mapping function between a workload w and an ACS y is a function
$\sigma : States(w) \rightarrow States(y) \quad |$
$\forall(\langle e_1, e_2, e_3\rangle) \quad | \quad e_1 \in \mathbb{S} \wedge e_2 \in \mathbb{S} \wedge e_3 \in \mathbb{U} :$
$\quad \langle e_1, e_2, e_3\rangle \in \sigma(s).matrix \iff$
$\quad \langle e_1, e_2, e_3\rangle \in s.auth \quad \vee$
$\quad s.belongs(e_1) = e_2 \wedge e_2 \neq unaffiliated \wedge e_3 = skolem$

```
sigma (sw: t_States_w): t_States_y =
(#
        matrix:= (LAMBDA (record: t_Matrix_Triplet):
                sw`auth(record) OR
                sw`belongs(record`e1) =
                        record`e2 AND
                        record`e2 /= unaffiliated AND
                        record`e3 = skolem
        )
#)
```

Figure 14: Implementation: state-mapping function

usual operator that denotes the definition of an anonymous function, as it's customary in functional paradigm and λ-calculus systems. The code is in figure 14.

3.4.3.2 Query-mapping

π, the query-mapping, if basically defined as a case-switch construct: if the workload query is an authorization one, it's true iff the corresponding ACS authorization query is part of the theory of the mapped state. If the workload query is an affiliation one, then it's true iff the matrix of the mapped ACS state contains the corresponding $skolem$ triplet.

The use of π will probably be clearer to the reader by recalling the definition of correctness, the only place where π actually becomes relevant to the properties of the system. The code is in figure 15.

3.4.3.3 Action-Mapping

α is by far the most complicated element of the implementation. The action-mapping is in this case realized using two helpers, one for each workload queries, in order to introduce some segmentation and keep things easier to understand. The code is not reported due to typographic reasons, please refer to the appendix. α simply calls upon the correct helper by adding the ACS matrix to the list of arguments, as the helpers need to use just that element of the ACS state (also because, in this particular case, the ACS state contains nothing else).

The helper for the *joinCoalition* labels iterates over the *newAuth* argument of the label. Note that, since we are working in a higher order logic specification system, it's strongly advised to define iterations as recursions, which is the meaning of the *RECURSIVE* keyword. Recursive definition also need to specify the strictly decreasing *MEASURE*, a function with natural numbers as domain which decreases with each recursive call, used to guarantee termination of the recursion, as prescribed by the Hoare method[14]. The base case is when the *newAuth* elements is empty: the resulting action-mapping is nothing but the empty sequence of ACS commands. Otherwise, for every *newAuth* element, we have two options: either the subject of the triplet in exam was already affiliated in the previous ACS state or not. In the case he was, we must first delete his previous affiliation by issuing

---

Let $\mathbb{U}$ the set of all strings.

Let $\mathbb{S} \subset \mathbb{U}$ the set of all usable strings.

Let *AUTH* and *MATRIX* the constructors of authorization and matrix queries of the ACS.

The query-mapping function between a workload w and an ACS y is a function

$\pi : Queries(w) \times powerset(Queries(y)) \rightarrow booleans \quad |$

$\forall(\langle e_1, e_2, e_3 \rangle) \quad | \quad e_1 \in \mathbb{S} \wedge e_2 \in \mathbb{S} \wedge e_3 \in \mathbb{U} :$

$\quad \pi(q, Q) \iff$

$\qquad q \in authorization\_queries \wedge AUTH(q.aut) \in Q \quad \vee$

$\qquad q \in affiliation\_queries \wedge ($

$\qquad\quad bel.e_2 \neq unaffiliated \wedge MATRIX(\langle bel.e_1, bel.e_2, skolem \rangle) \in Q \quad \vee$

$\qquad\quad bel.e_2 = unaffiliated \wedge \forall(org \in \mathbb{S}) : MATRIX(\langle bel.e_1, org, skolem \rangle) \notin Q$

```
pi (qw: t_Queries_w, ty: finite_set[t_Queries_y]): boolean =
CASES qw OF
        authCons_w(auth):
                ty(authCons_y(auth)),
        belongsCons_w(bel):
                IF (bel`e2 /= unaffiliated) THEN
                        ty(matrixCons_y((# e1:=bel`e1, e2:=bel`e2, e3:=skolem #)))
                ELSE
                        (FORALL (org: t_Strings):
                                NOT ty(matrixCons_y(
                                        (#
                                                e1:=bel`e1,
                                                e2:=org,
                                                e3:=skolem
                                        #)))
                        )
                ENDIF
ENDCASES
```

Figure 15: Implementation: query-mapping function

a *removeMatrix* command. In both cases, we then proceed to adding the triplet and the new affiliation to the ACS matrix, and we move on to the next *newAuth* element.

The helper for the *leaveCoalition* labels is slightly more convoluted. First of all, it iterates over the ACS matrix, and therefore it stores it twice in its arguments: the first, *original* is the whole matrix, while *matrix* is just the part of it that hasn't been iterated on yet. For each matrix element, we check if this has to be removed. This is true in one of two cases: either it's an affiliation record for the organization that we are removing, or it's an authorization record whose subject is, in the whole matrix, affiliated with the removed organization. Note that we have to look for affiliations in the whole matrix, as we might have already passed the affiliation record that we are looking for in the matrix used for the iteration. After this check is performed, we move on to the next record.

CHAPTER 4

PROOFS

In this chapter, I outline the proofs required for validating the correctness property. All the steps reported, except when explicitly pointed out, were verified in PVS using the internal prover system. Since the syntax of the prover is a bit complicated and out of the interest of this work, the proofs are here reported in a more human-readable format.

Recall from previous chapters that correctness is given as the logical conjunction of two sub-properties: query-mapping preservation and action-mapping preservation.

(Query-mapping preservation)

$$\forall(s \in States(W), \quad q \in Queries(W)):$$

$$q \in Th(s) \iff \pi(q, Th(\sigma(s))$$

$$\wedge$$

(Action-mapping preservation)

$$\forall(s \in States(W), l \in Labels(W):$$

$$\sigma(NEXT(s,l)) = TERMINAL(\sigma(s), \alpha(\sigma(s), l))$$

To prove correctness, I am going to prove first query-mapping preservation, and then action-mapping preservation. For the latter, I'll need a different sub-proof for the two workload label types, *joinCoalition* and *leaveCoalition*, due to their deeply different behavior.

4.1 Query-mapping Preservation

$$\forall(s \in States(W), \quad q \in Queries(W)):$$
$$q \in Th(s) \iff \pi(q, Th(\sigma(s))$$

Query-mapping preservation must hold for all workload states and all workload queries. In our case, we only have two types of workload queries: authorization and affiliation. Recall that queries, both workload and ACS ones, take a tuple as argument and returns true or false for a state according to the specification of the entailment relation, and the subsequent theory function. The code version of the entailment functions is reported in figure 16. We see that an authorization query is true in a workload state iff the argument tuple is part of the authorization policy of that state, while an affiliation query is true iff the tuple argument represents a pairing of the affiliation function. Recall the implementation elements σ and π (figure 17). We can see that σ fills the ACM matrix with all the authorization triplets directly, and the affiliation pairs for affiliated subjects are included appending the $skolem$ constant, while unaffiliated subjects are not reported. For π, the

```
ENTAILMENT_w : finite_set[t_Entailments_w] =
(LAMBDA (e: t_Entailments_w):
        CASES e`query OF
                authCons_w(aut):
                        member(aut, e`state`auth),
                belongsCons_w(bel):
                        e`state`belongs(bel`e1) = bel`e2
        ENDCASES
)

ENTAILMENT_y : finite_set[t_Entailments_y] =
(LAMBDA (e: t_Entailments_y):
        CASES e`query OF
                authCons_y(aut):
                        member(aut, e`state`matrix),
                matrixCons_y(mat):
                        member(mat,  e`state`matrix)
        ENDCASES
)
```

Figure 16: Comparison of workload and ACS entailment

queries are answered in a natural way, trying to resemble the definition of σ. The purpose of query-mapping preservation is to verify if this resemblance is correct or not.

4.1.1 Proof for Authorization Queries

Let's start to prove the property for the authorization queries. Expanding the various definitions we see that a workload authorization query is part of the theory of the workload state iff it's contained in the sequence representing the authorization policy. That is:

$$q \in Theory(s) \iff q.aut \in s.auth$$

```
sigma (sw: t_States_w): t_States_y =
(#
        matrix:= (LAMBDA (record: t_Matrix_Triplet):
                sw`auth(record) OR
                sw`belongs(record`e1) = record`e2 AND
                record`e2 /= unaffiliated AND
                record`e3 = skolem
        )
#)

pi (qw: t_Queries_w, ty: finite_set[t_Queries_y]): boolean =
CASES qw OF
        authCons_w(auth):
                ty(authCons_y(auth)),
        belongsCons_w(bel):
                IF (bel`e2 /= unaffiliated) THEN
                        ty(matrixCons_y((# e1:=bel`e1, e2:=bel`e2, e3:=skolem #)))
                ELSE
                        (FORALL (org: t_Strings):
                                NOT ty(matrixCons_y(
                                        (#
                                                e1:=bel`e1,
                                                e2:=org,
                                                e3:=skolem
                                        #)))
                        )
                ENDIF
ENDCASES
```

Figure 17: Query-mapping relevant functions

Where $q.aut$ is the argument of the query. Now, after applying the definition of σ, we see that the mapped ACS state matrix will contain all the authorization triplets of the workload state, plus all the affiliation couples and their $skolem$ identifiers. Applying also π to the definition of query-mapping preservation, we get:

$$q.aut \in s.auth \iff$$

$$q.aut \in s.auth$$

$$\vee$$

$$\langle q.aut.e1, q.aut.e2 \rangle \in s.belongs \wedge q.aut.e3 = skolem$$

Which seems, apparently, false. This is where the types system comes in help. Recall the code implementation of workload queries previously shown in figure 5. See that the type of the aut argument is $t\_TripleString$. Let's look at both the definition of the strings types and of the $skolem$ constant. $t\_Strings$ and $t\_Skolems$ are two disjoint sub-types of the $t\_ALL\_Strings$ type. $t\_TripleString$ is made of three $t\_Strings$ elements, while $skolem$ has type $t\_Skolems$. Therefore, since authorization queries can only be asked with $t\_TripleString$ arguments, it will never be the case that $q.aut.e3 = skolem$, as the types of $q.aut.e3$ and of $skolem$ are disjoint for all $q.aut$. Therefore, the second element of the disjunction in the right-hand side of the equality of the thesis will be identically false, making the proof trivial in the form $A \iff A$.

4.1.2 Proof for Affiliation Queries

Let's expand the various definitions. We see that a workload affiliation query is part of the theory of the workload state iff it represents a $\langle x, y\rangle$ pair in the workload affiliation function. That is:

$$q \in Theory(s) \iff s.belongs(q.bel.e1) = q.bel.e2$$

Where $q.bel$ is the argument of the query. Similarly as before, we apply the definitions of the implementation elements. Now if we let *affiliation_tuples* the set of all triplets $\langle sub, s.belongs(sub), skolem\rangle$ where $s.belongs(sub) \neq$ *unaffiliated*, we can rewrite the thesis as:

$$\begin{aligned}
&s.belongs(q.bel.e1) = q.bel.e2 \iff\\
&q.bel.e2 \neq \mathit{unaffiliated} \wedge\\
&\qquad \langle q.bel.e1, q.bel.e2, skolem\rangle \in (s.auth \cup \mathit{affiliation\_tuples})\\
&\vee\\
&q.bel.e2 = \mathit{unaffiliated} \wedge\\
&\qquad \forall (org : t\_Strings) :\\
&\qquad \langle q.bel.e1, org, skolem\rangle \notin (s.auth \cup \mathit{affiliation\_tuples})
\end{aligned}$$

Once again, the type of the $skolem$ constant becomes extremely important. Since $s.auth$ is composed of $t\_TripleString$ elements, the fields of which cannot ever be the $skolem$ constant as discussed in the previous subjection, the only relevant elements of the $(s.auth \cup affiliation\_tuples)$ set become the affiliation tuples. If the subject is affiliated with an organization, then his triplet will appear in the affiliation tuplets. If the subject is unaffiliated, then by the definition his triplet will not be in the affiliation tuples set. In both cases, the property holds.

This concludes the proof for query-mapping preservation. Notice how, even for such a trivial and apparently intuitive sub-property, the proofs are completely non-trivial, and actually unsolvable, without relying on strong typization and disjointedness of the generic string type in usable strings and keywords. The exploration of the intrinsic complexity given by strings is one of the core themes of ACEF, but their dissertation is beyond the scope of this work.

4.2 Action-mapping Preservation

$$\forall (s \in States(W), \quad l \in Labels(W)) :$$

$$\sigma(next(s,l)) = TERMINAL(\sigma(s), \alpha(\sigma(s), l))$$

Action-mapping preservation is the sub-property of correctness that assures that state-mapping and action-mapping are coherent, i.e. that the path of ACS actions corresponding to a workload action is “the right one”. Note that action-mapping preservation, and therefore correctness, only care about the terminal state reached by applying the sequence of ACS actions (as ACEF impose this sequence to be finite in length, a terminal state is always well-defined), but do not specify anything about the intermediate states crossed during the path. This relates more to another ACEF property, safety, but its description is out of the scope of interest of this work.

Action-mapping preservation must hold for all possible workload actions and states. Since we only have two workload action types, *joinCoalition* and *leaveCoalition*, which nature is deeply different, we are going to need two different proofs strategies. More specifically, we are going to always use induction as proving strategy, but the elements onto which we induct will be different.

4.2.1 Proof for *joinCoalition* Labels

The proof for *joinCoalition* labels is made by induction on the *newAuth* argument. The base case operates on a generic workload state by applying a *joinCoalition* label with an empty *newAuth* and a generic *org* . The inductive scheme is based on the fact that any generic *newAuth* , and therefore any generic *joinCoalition* label, being finite in length, can be reached by adding one authorization triplet at a time. Following the induction, we can therefore proof the property for all cases. Since the code notation can become a bit cumbersome in the proofs, we are going to use N as an alias for the transition function of both workload and ACS, T for the $TERMINAL_y$ function and, where clear from context, states and labels will be identified simply by s and l, dropping the suffix that distinguish between workload and ACS.

```
correctness_AM_JC_base: LEMMA
(FORALL (
         sw: t_States_w,
         lw: t_Labels_w
):
         joinCoalition_w?(lw) AND newAuth(lw) = empty_seq
         IMPLIES
         sigma(NEXT_w(sw, lw)) = TERMINAL_y(sigma(sw), alpha(sigma(sw), lw))
)
```

Figure 18: Action-mapping preservation, base case for JC labels

4.2.1.1 Base Case

The base case states that, for an empty *newAuth* and a generic *org* , the property holds:

$$\forall(s : workload\_states, l : workload\_labels) :$$

$$l = joinCoalition(org, \emptyset)$$

$$\Rightarrow$$

$$\sigma(N(s,l)) = T(\sigma(s), \alpha(\sigma(s), l)))$$

The code version of this property is given in figure 18 In order to proceed, we first expand the definitions of the α function and of $TERMINAL_y$. Keeping in mind the premises and removing them for ease of reading, we obtain $\sigma(N(s,l)) = \sigma(s)$. By applying the transition function, and dropping the σ from both sides, we obtain two sub-clauses:

```
$s.auth = s.auth \cup l.newAuth$
$\wedge$
$s.belongs =$
$\lambda(sub : t\_Strings) :$
    $IF \quad \exists(x : t\_TripleString) : (x`e1 = sub \wedge member?(newAuth, x)) \quad THEN$
        $org$
    $ELSE$
        $s`belongs(sub)$
    $ENDIF$
```

Since *newAuth* is empty, the first part of the main conjunction is trivially true.

For the second part of the conjunction, we define that equality between functions hold iff their domain and co-domains are the same and, for all elements of the domain the two functions return the same element of the co-domain. Since *newAuth* is empty, the condition of the *IF* is always false, and therefore the *ELSE* branch is taken for all cases. Therefore, the second sub-clause is true. This concludes the proof for the base case.

```
correctness_AM_JC_step: LEMMA
(FORALL (
        sw: t_States_w,
        lw2: t_Labels_w,
        lw1: t_Labels_w,
        na: t_TripleString
):
        joinCoalition_w?(lw1) AND
        joinCoalition_w?(lw2) AND
        org(lw1) = org(lw2) AND
        newAuth(lw1) = allButLast(newAuth(lw2)) AND
        newAuth(lw2) = concat(newAuth(lw1), sequence(na)) AND

        sigma (NEXT_w(sw, lw1)) = TERMINAL_y( sigma(sw), alpha(sigma(sw), lw1))
        IMPLIES
        sigma(NEXT_w(sw, lw2)) = TERMINAL_y( sigma(sw), alpha(sigma(sw), lw2))
)
```

Figure 19: Action-mapping preservation, induction step for JC labels

4.2.1.2 Inductive Step

The code for the inductive step is given in figure 19. Informally speaking, we want to proof that if the property holds for a *joinCoalition* label with *newAuth* of length n, this implies that it will hold for the same $newAuth$ when we append to it an additional na triplet, reaching length $n + 1$. In other words:

$$\begin{aligned}
&l_2.newAuth = l_1.newAuth \bullet na \quad \wedge \\
&l_2.org = l_1.org \quad \wedge \\
&\sigma(N(s, l_1)) = T(\sigma(s), \alpha(\sigma(s), l_1)) \\
&\Rightarrow \\
&\sigma(N(s, l_2)) = T(\sigma(s), \alpha(\sigma(s), l_2))
\end{aligned}$$

To do this, consider the nature of the $TERMINAL_y$ function. What it does is recursively applying the first label of the sequence provided as argument starting from an initial state, and eventually reaching a final state. It's easy to see that if we define an operator sub that extracts sub-sequences from a sequence, taking the 0-based indexes of the starting and final elements of the sub-sequence, the following property holds:

$$\begin{aligned}
&\forall k \quad | \quad 0 \leq k < length(\alpha(s, l)) : \\
&T(s, \alpha(s, l)) = \\
&T(T(s, sub(\alpha(s, l), 0, k), sub(\alpha(s, l), k, length(\alpha(s, l))))
\end{aligned}$$

Basically, we are saying that the terminal state reached is the same by walking the path all in one go or by stopping in one of the intermediate states and then continuing. Applying this property to the thesis equation, after some manipulation, we get:

$$\sigma(N(s,l_1)) = T(\sigma(s), \alpha(\sigma(s), l_1))$$

$$\Rightarrow$$

$$\sigma(N(s,l_2)) =$$

$$T(T(s,\alpha(s,l_1)), \alpha(T(s,\alpha(s,l_1)), l_d))$$

Where l_d is a fictitious label with a *newAuth* composed only of the na triplet. This is a consequence of the fact that α is made in such a way to iterate always in the same order over the *newAuth* elements. Having appended na, the first part of the resulting ACS actions sequence will be identical, and the remaining 3 elements, according to the specific case of na, will be new.

According to this implementation of the α helper, if the subject was unaffiliated in the previous state the two last elements will be the adding of the new affiliation and the adding of the new authorization triplet, while if the subject was already affiliated an additional removal of the old affiliation will be issued before the other two elements. If we now apply the inductive hypothesis, we obtain:

$$\sigma(N(s,l_2)) = T(\sigma(N(s,l_1)), \alpha(\sigma(N(s,l_1)), l_d)$$

```
alpha_helpJC (original: t_Matrix_y, org: t_Strings, newAuth: t_Seq[t_TripleString]):
RECURSIVE {out: t_Seq[t_Labels_y] | out`length = 3 * newAuth`length} =
IF newAuth`length=0 THEN
        empty_seq
ELSEIF
(EXISTS (oldOrg: t_Strings):
original((# e1:= first(newAuth)`e1, e2:= oldOrg, e3:=skolem#))) THEN
        append( append( append( alpha_helpJC(original, org, rest(newAuth)),
                % Remove old affiliation
                removeMatrixCons_y(
                (#
                        e1:= first(newAuth)`e1,
                        e2:= choose({oldOrg: t_Strings |
                                original(
                                (#
                                        e1:= first(newAuth)`e1,
                                        e2:= oldOrg,
                                        e3:= skolem
                                #))}),
                        e3:= skolem
                #))),

                % Add new affiliation
                addMatrixCons_y(
                (# e1:= first(newAuth)`e1, e2 := org, e3 := skolem #))),

                % Add the new right
                addMatrixCons_y(first(newAuth)))
ELSE
        append( append( alpha_helpJC(original, org, rest(newAuth)),
                % Add new affiliation
                addMatrixCons_y(
                (#
                        e1:= first(newAuth)`e1,
                        e2 := org,
                        e3 := skolem
                #))),
                        % Add the new right
                addMatrixCons_y(first(newAuth)))
ENDIF
MEASURE newAuth`length
```

Figure 20: Action-mapping, helper for JC labels

Expanding two or three times T and expanding the corresponding N we obtain

$$\begin{aligned}
&\sigma(N(s, l_2)).matrix = \\
&\sigma(N(s, l_1)).matrix \\
&\setminus \langle na.e1, oldOrg, skolem\rangle \\
&\cup \langle na.e1, l_1.org, skolem\rangle \\
&\cup\, na
\end{aligned}$$

Where $oldOrg$ is the organization to which the subject of na is affiliated in the state $\sigma(N(s, l_1))$, if such an organization exists. Note that the triplet $\langle na.e1, oldOrg, skolem\rangle$ might not be part of the state, as $na.e1$ may be unaffiliated. Nevertheless, the use of sets difference is still valid. Proceeding in the expansion, we obtain an equality between two finite sets, the matrices of the ACS states, i.e. of their matrices. Recall that sets are equivalent to their characteristic functions, and therefore we have an equality between boolean functions (predicates). Generalizing over the argument string, we obtain a logical equality over the generic triplet r of the type:

$$
\begin{aligned}
& s.auth(r) \quad \vee \\
& l_2.newAuth(r) \quad \vee \\
& N(s, l_2).belongs(sub) \neq \mathit{unaffiliated} \wedge r = \langle sub, N(s, l_2).belongs(sub), skolem\rangle \\
& \iff \\
& s.auth(r) \quad \vee \\
& l_1.newAuth(r) \quad \vee \\
& N(s, l_1).belongs(sub) \neq \mathit{unaffiliated} \wedge r = \langle sub, N(s, l_1).belongs(sub), skolem\rangle \wedge sub \neq na.e1 \quad \vee \\
& r = \langle na.e1, l_d.org, skolem\rangle \quad \vee \\
& r = na
\end{aligned}
$$

After some further logical simplification, and using the same typing tricks used for proving query-mapping preservation ($skolem$ can't be in authorization triplets), and changing the order of the elements we obtain:

$$l_2.newAuth(r) \quad \vee$$

$$N(s,l_2).belongs(sub) \neq \mathit{unaffiliated} \wedge r = \langle sub, N(s,l_2).belongs(sub), skolem \rangle$$

$$\iff$$

$$(l_1.newAuth(r) \vee r = na) \quad \vee$$

$$N(s,l_1).belongs(sub) \neq \mathit{unaffiliated} \wedge r = \langle sub, N(s,l_1).belongs(sub), skolem \rangle \wedge sub \neq na.e1 \quad \vee$$

$$r = \langle na.e1, l_d.org, skolem \rangle$$

We can see that, for what concerns the authorization part, the equality holds: being part of $l_2.newAuth$ means either being part $l_1.newAuth$ or being na. For what concerns affiliation, things are slightly less straight-forward: we need to separate the proof for the subject in the na triplet from the others. The former will have its affiliation changed, possibly with the same organization if he already appeared in one of the $l_1.newAuth$ triplets, in the application of l_d, while the others will keep the affiliation obtained by applying l_1.

For these latter subjects. the property rewrites as:

$$N(s, l_2).belongs(sub) \neq \mathit{unaffiliated} \wedge r = \langle sub, N(s, l_2).belongs(sub), skolem \rangle$$

$$\iff$$

$$N(s, l_1).belongs(sub) \neq \mathit{unaffiliated} \wedge r = \langle sub, N(s, l_1).belongs(sub), skolem \rangle$$

Which is true, as their affiliation doesn't change during l_d, and therefore their belongs function is the same in both $N(s, l_1).belongs(sub)$ and $N(s, l_2).belongs(sub)$

For the subject of na we know that his new affiliation is, no matter the previous one, $l_d.org$, as we remove his old affiliation and replace it. Since *org* is the same for all l_2, l_1 and l_d, and *org* cannot be the *unaffiliated* element, the re-writing becomes:

$$r = \langle sub, org, skolem \rangle$$

$$\iff$$

$$r = \langle sub, org, skolem \rangle$$

Which is trivially true.

Having proved the base case and the induction step, we can now apply induction over the length of the *newAuth* argument. This concludes the proof of action-mapping preservation for *joinCoalition* labels.

4.2.2 Proof for *leaveCoalition* labels

leaveCoalition labels require a different approach. Since their argument is one single *org* string, we cannot induct on their argument. Rather, we are going to induct on the ACS state itself. It's important to notice that an "empty" matrix of the ACS state represent a workload state where nothing is allowed by the authorization policy and all the subjects are unaffiliated. From this state, we can add, one at a time, triplets that either represent one authorization triplet or change one subject-organization pair in the workload affiliation function, until we reach the generic $\sigma(s)$ state for which we want to prove our property. Due to technique used, the sub-proof for *leaveCoalition* labels hasn't been implemented in PVS. The motivation behind this decision will be clearer after exposing the rationale behind the base case.

4.2.2.1 Base Case

In the base case, $\sigma(s_w).matrix$ is an empty set.

The helper for α, being $\sigma(s_w).matrix$ empty, returns an empty list of ACS commands. Therefore, $TERMINAL_y$ leaves the state unchanged, and by expanding, manipulating and recalling that for ACS states, being them composed only of the matrix, the equality of states is the same as the equality between matrices, we obtain $\emptyset = \emptyset$, which is trivially true. As mentioned, we can expand this basic ACS state matrix via either adding an authorization triplet, or via adding the triplet that represents an affiliation pair. Let's analyze these two induction steps separately.

4.2.2.2 Induction Step 1: Adding an Authorization Triplet

Let's call s the generic workload state and s' the generic workload state with authorization policy given by $na \bullet s.auth$, where na is an authorization triplet, and $\bullet$ is the concatenation operator. I decided to expand the sequence by pre-pending the element, but this isn't affecting the proof strategy, only the cumbersomeness of the notation. The thesis to prove is:

$$\sigma(N(s, LC(org)) = T(\sigma(s), \alpha(\sigma(s), LC(org)))$$

$$\Rightarrow$$

$$\sigma(N(s', LC(org)) = T(\sigma(s'), \alpha(\sigma(s'), LC(org)))$$

Since ACS state are composed only of the matrix, their equality is the same as the equality between their matrices. Given this consideration, recalling that matrices are set and therefore predicates, we can rewrite $\sigma(s')$ as a predicate over a generic triplet r:

$$\begin{aligned}&r = na \quad \vee\\ &r \in s.auth \quad \vee\\ &s'.belongs(r.e1) = r.e2 \quad \wedge\\ &r.e2 \neq unaffiliated \quad \wedge\\ &r.e3 = skolem\end{aligned}$$

and $\alpha(\sigma(s'), LC(org))$ becomes $\alpha' \bullet \alpha(\sigma(s), LC(org))$ where α' is different according to whether we have expanded using a na triplet where the subject is affiliated with *org* (and therefore we need to remove this newly assigned right) or if he's affiliated with an organization different from *org* .

$$\begin{aligned}&\sigma(N(s, LC(org)) = T(\sigma(s), \alpha(\sigma(s), LC(org)))\\ &\Rightarrow\\ &\sigma(N(s', LC(org)) = T(\sigma(s'), \alpha' \bullet \alpha(\sigma(s), LC(org)))\end{aligned}$$

The elements in $\alpha(\sigma(s), LC(org))$ are in the exact same order, as the action-mapping helper is deterministic. As you can see, the helper checks the original ACS matrix only w.r.t. affil-

```
alpha_helpLC(original: t_Matrix_y, org: t_Strings, matrix: t_Matrix_y):
RECURSIVE t_Seq[t_Labels_y] =
        IF card(matrix) = 0 THEN
                empty_seq
        ELSIF choose(matrix)`e2 = org THEN
                append(alpha_helpLC(original, org, rest(matrix)),
                        removeMatrixCons_y(choose(matrix)))
        ELSIF original((# e1:= choose(matrix)`e1, e2:= org, e3:= skolem #)) THEN
                append(alpha_helpLC(original, org, rest(matrix)),
                        removeMatrixCons_y(choose(matrix)))
        ELSE
                alpha_helpLC(original, org, rest(matrix))
        ENDIF
MEASURE card(matrix)
```

Figure 21: Action-mapping preservation, helper for LC labels

iation triplets, and therefore, having expanded the matrix adding an authorization triplet, the result produced by α will be the same w.r.t. affiliation triplets and authorization triplets different from na.

Sub-case 1: *na.e1* is affiliated with *org*

In this case, when examining the na element, the helper will issue a removal of na, entering the second $ELSIF$ branch. Therefore, we are going to have $\alpha' = removeMatrix(na)$. Under this assumption, we obtain:

$$\sigma(N(s, LC(org)) = T(\sigma(s), \alpha(\sigma(s), LC(org)))$$

$$\Rightarrow$$

$$\sigma(N(s', LC(org)) = T(\sigma(s), \alpha(\sigma(s), LC(org)))$$

This is because the only difference between $\sigma(s)$ and $\sigma(s')$ is the presence of na, which gets removed by the first action issued by the helper. This is coherent with the expected behavior: since we are adding a na that we need to immediately void, the terminal state will be the same as never having added na in the first place. Rewriting the succedent using the antecedent and expanding σ and $NEXT$, we obtain:

$$\begin{aligned}
&s'.belongs(r.e1) \neq org \land (r = na \lor r \in s.auth) \quad \lor \\
&s'.belongs(r.e1) \neq org \land (s'.belongs(r.e1) = r.e2 \land r.e2 \neq unaffiliated \land r.e3 = skolem) \\
&\iff \\
&s.belongs(r.e1) \neq org \land r \in s.auth \quad \lor \\
&s.belongs(r.e1) \neq org \land (s.belongs(r.e1) = r.e2 \land r.e2 \neq unaffiliated \land r.e3 = skolem)
\end{aligned}$$

Note that $s.belongs = s'.belongs$ as the only difference is the na authorization triplet. Since we are under the assumption that $s.belongs(na.e1) = org$, if $r = na$ then the first line is identically false, guaranteeing the logical equivalence and proving the induction step.

Sub-case 2: *na.e1* is not affiliated with *org*

In this case, when examining the na element, the helper will not generate any ACS action, ending in the $ELSE$ branch of the switch. Therefore, $\alpha' = empty\_seq$. This is coherent with the expected behaviour: since the helper for LC only issues removals, and since na has to remain untouched by the application of LC as its subject isn't affiliated to org, no action must be taken w.r.t. na. Since $\alpha' = empty\_seq$, we can rewrite our thesis in:

$$\sigma(N(s, LC(org)) = T(\sigma(s), \alpha(\sigma(s), LC(org)))$$

$$\Rightarrow$$

$$\sigma(N(s', LC(org)) = T(\sigma(s'), \alpha(\sigma(s), LC(org)))$$

Since na isn't part of s, it won't be part of $\sigma(s)$ and therefore $\alpha(\sigma(s), LC(org))$ will not issue its removal. Therefore, na will still be there at the end of all the path traversal of T, and we can rewrite the whole succedent as:

$$\sigma(N(s', LC(org)).matrix = T(\sigma(s), \alpha(\sigma(s), LC(org))).matrix \cup na$$

Which becomes, using the antecedent of the thesis

$$\sigma(N(s', LC(org)).matrix = \sigma(N(s, LC(org)).matrix \cup na$$

Which is then proven directly by expanding σ and N, or informally by noticing that s' and s only differ from na and that this is directly translated in the ACS matrix.

4.2.2.3 Induction Step 2: Adding an Affiliation Pair

First of all, recall that we are now expanding the state by modifying the affiliation of one unaffiliated subject with an new subject-organization pair. Suppose now to have a workload state s. We are now expanding to a state s', almost identical to s but where the affiliation function of a specific subject,

$$\sigma(N(s, LC(org)) = T(\sigma(s), \alpha(\sigma(s), LC(org)))$$

$$\Rightarrow$$

$$\sigma(N(s', LC(org)) = T(\sigma(s'), \alpha(\sigma(s'), LC(org)))$$

For ease of reading, we are going to define $B_x = \langle newSub, newOrg, skolem\rangle$, i.e. the matrix triplet corresponding to our changed affiliation. Note that, according to the definition of σ, unaffiliated subjects do not have a corresponding triplet matrix. Therefore, our ACS matrices of the two states will differ only by the presence of B_x. Similarly as before, we have two different sub-cases, given by $newOrg \neq org$ and $newOrg = org$.

Sub-case 1: *newOrg <> org*

In this sub-case, we get $\alpha(\sigma(s'), LC(org)) = \alpha(\sigma(s), LC(org))$. This is because the helper is only affected by affiliation records of the removed organization, and B_x isn't one. Therefore, the removal of B_x will not be issued by the helper and we can rewrite the sequent of the thesis, also by applying the inductive hypothesis, as:

$$\sigma(N(s', LC(org)).matrix = \sigma(N(s, LC(org)).matrix \cup B_x$$

Which is then proven by expanding σ and N.

Sub-case 1: *newOrg = org*

This is the trickiest case. Recall that α, for LC labels, issues only removals, and it issues the removal of all the rights of subjects affiliated with the removed organization, as well as the removal of their affiliation (as non-affiliation is represented by the lack of an affiliation triplet in the ACS matrix).

Therefore, $\alpha(\sigma(s'), LC(org))$ will contain all the elements of $\alpha(\sigma(s), LC(org))$, together with the removals of all the rights of $newSub$. But we don't know in which order these elements will appear interleaved in the elements of $\alpha(\sigma(s), LC(org))$. Nevertheless, we know from transaction theory that since the terminal state reached after applying a sequence of commands all with the same "polarity", i.e. all removal or all addition, will be the same no matter the order in which the commands are executed. Since all the elements are removal, we can ignore the order, and rewrite into:

$$\sigma(N(s', LC(org)) =$$

$$T(\sigma(s'), \alpha(\sigma(s'), LC(org)) \bullet \alpha' \bullet removeMatrix(B_x))$$

as we also need to remove B_x. α' is the sequence of removals of all the rights of $newSub$, which might also be empty.

Utilizing the same property of graph-paths used already in the previous subsection, expanding and rewriting, we obtain:

$$\sigma(N(s', LC(org)) = T(\sigma(s), \alpha')$$

Which is coherent with the expected behavior: the reached state is such that all the rights of $newSub$ are voided, and $newSub$ is back to be unaffiliated, as it was in state s. A more formal proof can be obtained by expanding the definitions.

4.3 Summary

In this chapter we have seen how to prove correctness for the case study in analysis. First, we have proven query-mapping preservation for all the possible queries. After that, we have proven action-mapping preservation for the two possible labels, each one of which needed two different sub-cases.

What is interesting to see is how strong typization and a precise hierarchy definition was necessary to achieve the proof of query-mapping preservation, and how action-mapping preservation strongly depends on some sort of induction to achieve its proof.

CHAPTER 5

CONCLUSIONS

This thesis presented a sub-set of the ACEF theoretical framework. ACEF is a novel tool that allows to analyze the implementation of authorization policies of generic applications, together with all the data required by the application, within a generic ACS. ACEF offers the definition of some relevant properties of these implementations, the most important of which is correctness. The formal proof of these properties is usually a convoluted process, full of sub-cases, and prone to overlooking of the details. To assist the analysts in this task, I developed a PVS structure to hold ACEF case studies. PVS is a formal prover system that was chosen because of its higher order logic capabilities. The backing of a formal prover system guarantees that no detail is overlooked by the analysts, and that the final results are correct. To show the capabilities of this approach, I proved correctness for the "dynamic coalitions" scenario, one of the case studies behind the birth of ACEF.

There are some major hindsight obtained from the specific case study analyzed. First of all, a strong typing system helps a lot during the proof of the query-mapping preservation sub-property of correctness. This is, in my opinion, not a general result, but rather due to the fact that the specific implementation in analysis relied on the existence of reserved keywords, such as the $skolem$ constants: if ACEF had been used to its full extent, the implementation would also have had to take care of other relevant properties such as homomorphism and compatibility of the implementation, properties that are based on the concept that strings do not hold sub-types. Therefore, in that case, it’s hard to imagine that type information becomes determinant in the proofs. Nevertheless, the implementation we analyzed is a legitimate one, as most real-world system allow for the use and definition of reserved keywords.

Another noticeable aspect is related to the fact that the proofs for action-mapping preservation always depend on some sort of inductive strategy. This seems, in my opinion, a result that is probably general: action-mapping works with sequences of commands, and when working with these kinds of data structures induction usually seems the most fitting strategy. Note that, albeit I developed the proofs by inducting on various elements of the theory, I was actually always inducting on the length of the sequence returned from the α function, and was just splitting down the possible causes of its expansion to maintain clarity in the exposition of an intrinsically complicated and branching proof.

The final consideration is the fact that, even for a simple scenario like the one analyzed, proofs can become quite complicated. They are never conceptually difficult to grasp, but they present lots of sub-cases and an high level of detail and depth. This is why I believe that the integration between ACEF and automated proving systems is a worthy effort: the development of these proofs by hand leaves too many details on the shoulders of the analyst, and the human brain is much better at thinking of the big picture rather than taking care of the details, a task much more suited for the machine. That being said, it’s hard to imagine the development of a completely automated tool that could go beyond proving the trivially simplest implementations. A computer-assisted proving system seems a far more realistic final outcome of this development path.

For the future work on this topic, I plan on improving the current PVS place-holder in order to be more flexible towards generic scenarios, and possibly verify if the hindsight obtained from the case study really are general and if they can be integrated in the deductive process. Finally, if possible, it would be interesting to develop a library-like PVS file that contains the most useful lemmas that seem to often "pop-up" during the proofs.

APPENDICES

Appendix A

CODE FOR ACEF DATA STRUCTURE

```
ACEF_data_structure[T: TYPE]: THEORY
BEGIN

t_Seq: TYPE =
[#
        length: nat,
        seq: {s: [below[length]->T] |
        (FORALL (n1: below[length], n2: below[length]):
        n1 /= n2 IFF s(n1) /= s(n2))}
#]

empty_seq: t_Seq =
(#
        length := 0,
        seq := (LAMBDA (x: below[0]): epsilon! (t:T): true)
#)

first(a: {aa: t_Seq | aa`length > 0}): T = a`seq(0)

rest(a: {aa: t_Seq | aa`length>0}): t_Seq =
(#
        length:= a`length-1,
        seq:= (LAMBDA (n:below[a`length-1]): a`seq(n+1))
#)

last(a: {aa: t_Seq | aa`length > 0}): T = a`seq(a`length-1)

member?(a: t_Seq, x: T): bool =
        (EXISTS (i: below[a`length]): a`seq(i) = x)

disjoint?(a: t_Seq, b: t_Seq): bool =
        (FORALL (i: below[b`length]): NOT member?(a, b`seq(i)))

prepend(a: t_Seq, x: {xx: T | NOT member?(a, xx)}): t_Seq =
(#
        length:= a`length+1,
        seq:= (LAMBDA (i: below[a`length+1]):
                IF  i=0 THEN
                        x
                ELSE
                        a`seq(i-1)
                ENDIF)
#)

concat(a: t_Seq, b: {bb: t_Seq | disjoint?(a, bb)}): t_Seq =
(#
        length:= a‘length + b‘length,
        seq:= (LAMBDA (i: below[a‘length + b‘length]):
                IF i<a‘length THEN
                        a‘seq(i)
                ELSE
                        b‘seq(i − a‘length)
                ENDIF)
#)

position(a: t_Seq, x: {xx: T | member?(a, xx)}):
RECURSIVE below[a‘length] =
        IF (first(a) = x) THEN
                0
        ELSE
                1+position(rest(a), x)
        ENDIF
MEASURE a‘length

remove(a: t_Seq, x: T): t_Seq =
        IF member?(a, x) THEN
                (#
                        length:= a‘length −1,
                        seq:= (LAMBDA (i: below[a‘length −1]):
                                IF i < position(a, x) THEN
                                        a‘seq(i)
                                ELSE
                                        a‘seq(i+1)
                                ENDIF)
                #)
        ELSE
                a
        ENDIF

toFiniteSet(a: t_Seq): finite_set[T] =
        (LAMBDA(x: T): member?(a, x))

END ACEF_data_structure
```

Appendix B

PVS CODE FOR THE CASE STUDY

```
coalition_AMa: THEORY
BEGIN
IMPORTING finite_sets
IMPORTING ACEF_data_structure

t_ALL_Strings: TYPE+
t_Strings_pred: [t_ALL_Strings -> bool]
t_Strings: TYPE+ = (t_Strings_pred)
t_Skolems_pred: [t_ALL_Strings -> bool]
t_Skolems: TYPE+ = (t_Skolems_pred)
Strings_Nature_Axiom: AXIOM
(
        FORALL (s: t_ALL_Strings): t_Strings_pred(s) IFF NOT t_Skolems_pred(s)
)

t_DoubleString: TYPE =
[#
        e1 : t_Strings,
        e2 : t_Strings
#]

t_TripleString: TYPE =
[#
        e1 : t_Strings,
        e2 : t_Strings,
        e3 : t_Strings
#]

t_Matrix_Triplet: TYPE =
[#
        e1: t_Strings,
        e2: t_Strings,
        e3: t_ALL_Strings
#]

t_Subjects_w : TYPE = finite_set [t_Strings]
t_Objects_w :  TYPE = finite_set [t_Strings]
t_Rights_w : TYPE = finite_set [t_Strings]
t_Auth_w : TYPE = finite_set [t_TripleString]
t_Organizations_w : TYPE = finite_set [t_Strings]
t_Belongs_w : TYPE = [t_Strings -> t_Strings]
```

Appendix B (Continued)

```
t_States_w : TYPE =
[#
        subjects:       t_Subjects_w,
        objects:        {obj: t_Objects_w | disjoint?(obj, subjects)},
        rights:         {rig: t_Rights_w |
                                        disjoint?(rig, objects) AND
                                        disjoint?(rig, subjects)},
        auth:           {aut: t_Auth_w | (FORALL (a: t_TripleString):
                                        member(a, aut) IMPLIES
                                        member(a`e1, subjects) AND
                                        member(a`e2, objects) AND
                                        member(a`e3, rights))},
        organizations:  {org: t_Organizations_w |
                                                disjoint?(org, rights) AND
                                                disjoint?(org, objects) AND
                                                disjoint?(org, subjects) AND
                                                member(unaffiliated, org)},
        belongs:        {bel: t_Belongs_w | (FORALL (b: t_Strings):
                                        IF member(b, subjects) THEN
                                        member(bel(b), organizations)
                                        ELSE
                                                bel(b) = invalid
                                        ENDIF)}
#]

t_Queries_w : DATATYPE
BEGIN
        authCons_w(
                aut: t_TripleString
        ): auth_w?

        belongsCons_w(
                bel: t_DoubleString
        ): belongs_w?
END t_Queries_w

t_Labels_w : DATATYPE
BEGIN
        joinCoalitionCons_w(
                org     : t_Strings,
                newAuth : t_Seq [t_TripleString]
        ): joinCoalition_w?

        leaveCoalitionCons_w(
                org     : t_Strings
        ): leaveCoalition_w?
```

```
END t_Labels_w

t_Entailments_w : TYPE =
[#
        state : t_States_w,
        query : t_Queries_w
#]

ENTAILMENT_w : finite_set[t_Entailments_w] =
(LAMBDA (e: t_Entailments_w):
        CASES e`query OF
                authCons_w(aut): member(aut, e`state`auth),
                belongsCons_w(bel): e`state`belongs(bel`e1) = bel`e2
        ENDCASES
)

THEORY_w(state_w : t_States_w) : finite_set[t_Queries_w] =
(LAMBDA (query_w : t_Queries_w) :
        ENTAILMENT_w((# state:=state_w, query:=query_w #))
)

NEXT_w (state: t_States_w, label: t_Labels_w) : t_States_w =
CASES label OF
        joinCoalitionCons_w(org,newAuth):
        (#
                subjects := state`subjects,
                objects := state`objects,
                rights := state`rights,
                auth := union(state`auth, toFiniteSet(newAuth)),
                organizations := state`organizations,
                belongs :=
                        (LAMBDA (s: t_Strings):
                        IF (EXISTS (x: t_TripleString): x`e1 = s AND
                        member?(newAuth, x)) THEN
                                org
                        ELSE
                                state`belongs(s)
                        ENDIF)
        #),
        leaveCoalitionCons_w(org):
        (#
                subjects := state`subjects,
                objects := state`objects,
                rights := state`rights,
                auth :=
                        (LAMBDA (x: t_TripleString):
                        state`auth(x) AND state`belongs(x`e1) /= org),
                organizations := state`organizations,
                belongs :=
                        (LAMBDA (s: t_Strings):
                        IF state`belongs(s) = org THEN
                                unaffiliated
                        ELSE
                                state`belongs(s)
                        ENDIF)
        #)
ENDCASES

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

t_Matrix_Triplet: TYPE =
[#
        e1: t_Strings,
        e2: t_Strings,
        e3: t_ALL_Strings
#]

t_Matrix_y : TYPE = finite_set[t_Matrix_Triplet]

t_States_y : TYPE =
[#
        matrix: t_Matrix_y
#]

t_Queries_y : DATATYPE
BEGIN
        authCons_y(
                aut : t_TripleString
        ): auth_y?

        matrixCons_y(
                mat : t_Matrix_Triplet
        ): matrix_y?
END t_Queries_y

t_Labels_y : DATATYPE
BEGIN
        addMatrixCons_y(
                mat : t_Matrix_Triplet
        ): addMatrix_y?

        removeMatrixCons_y(
                mat : t_Matrix_Triplet
        ): removeCoalition_y?
```

Appendix B (Continued)

```
END t_Labels_y

t_Entailments_y : TYPE = [# state: t_States_y, query: t_Queries_y #]

ENTAILMENT_y : finite_set[t_Entailments_y] =
(LAMBDA (e: t_Entailments_y):
        CASES e`query OF
                authCons_y(aut):
                        member(aut, e`state`matrix),
                matrixCons_y(mat):
                        member(mat,  e`state`matrix)
        ENDCASES
)

THEORY_y(sy : t_States_y) : finite_set[t_Queries_y] =
(LAMBDA (qy : t_Queries_y) :
        ENTAILMENT_y((# state:=sy, query:=qy #))
)

NEXT_y (state: t_States_y, label: t_Labels_y) : t_States_y =
CASES label OF
        addMatrixCons_y(mat):
                (#
                        matrix := add(mat, state`matrix)
                #),
        removeMatrixCons_y(mat):
                (#
                        matrix := remove(mat, state`matrix)
                #)
ENDCASES

TERMINAL_y (sy: t_States_y, ly_star: t_Seq[t_Labels_y]):
RECURSIVE t_States_y =
        IF ly_star`length = 0 THEN
                sy
        ELSE
                NEXT_y(TERMINAL_y(sy, allButLast(ly_star)),last(ly_star))
        ENDIF
MEASURE ly_star`length

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

alpha_helpJC (original: t_Matrix_y, org: t_Strings, newAuth: t_Seq[t_TripleString]):
RECURSIVE {out: t_Seq[t_Labels_y] | out`length = 3 * newAuth`length} =
IF newAuth`length=0 THEN
        empty_seq
ELSEIF
```

Appendix B (Continued)

```
(EXISTS (oldOrg: t_Strings):
original((# e1:= first(newAuth)‘e1, e2:= oldOrg, e3:=skolem#))) THEN
        append( append( append( alpha_helpJC(original, org, rest(newAuth)),
                % Remove old affiliation
                removeMatrixCons_y(
                (#
                        e1:= first(newAuth)‘e1,
                        e2:= choose({oldOrg: t_Strings |
                                original(
                                (#
                                        e1:= first(newAuth)‘e1,
                                        e2:= oldOrg,
                                        e3:= skolem
                                #))}),
                        e3:= skolem
                #))),

                % Add new affiliation
                addMatrixCons_y(
                (# e1:= first(newAuth)‘e1, e2 := org, e3 := skolem #))),

                % Add the new right
                addMatrixCons_y(first(newAuth)))
ELSE
        append( append( alpha_helpJC(original, org, rest(newAuth)),
                % Add new affiliation
                addMatrixCons_y(
                (#
                        e1:= first(newAuth)‘e1,
                        e2 := org,
                        e3 := skolem
                #))),
                        % Add the new right
                addMatrixCons_y(first(newAuth)))
ENDIF
MEASURE newAuth‘length

alpha_helpLC(original: t_Matrix_y, org: t_Strings, matrix: t_Matrix_y):
RECURSIVE t_Seq[t_Labels_y] =
        IF card(matrix) = 0 THEN
                empty_seq
        ELSIF choose(matrix)‘e2 = org THEN
                append(alpha_helpLC(original, org, rest(matrix)),
                        removeMatrixCons_y(choose(matrix)))
        ELSIF original((# e1:= choose(matrix)‘e1, e2:= org, e3:= skolem #)) THEN
                append(alpha_helpLC(original, org, rest(matrix)),
                        removeMatrixCons_y(choose(matrix)))
        ELSE
                alpha_helpLC(original, org, rest(matrix))
        ENDIF
MEASURE card(matrix)

alfa (sy: t_States_y, lw : t_Labels_w): t_Label_STAR_y =
CASES lw OF
      joinCoalitionCons_w(org,newAuth): alfa_helper_joinCoalitionCons_w(org, newAuth)
      leaveCoalitionCons_w(org): alfa_helper_leaveCoalitionCons_w(org, sy`matrix)
ENDCASES

sigma (sw: t_States_w): t_States_y =
(#
        matrix:= (LAMBDA (record: t_Matrix_Triplet):
                sw`auth(record) OR
                sw`belongs(record`e1) =
                        record`e2 AND
                        record`e2 /= unaffiliated AND
                        record`e3 = skolem
        )
#)

pi (qw: t_Queries_w, ty: finite_set[t_Queries_y]): boolean =
CASES qw OF
        authCons_w(auth):
                ty(authCons_y(auth)),
        belongsCons_w(bel):
                IF (bel`e2 /= unaffiliated) THEN
                        ty(matrixCons_y((# e1:=bel`e1, e2:=bel`e2, e3:=skolem #)))
                ELSE
                        (FORALL (org: t_Strings):
                                NOT ty(matrixCons_y(
                                        (#
                                                e1:=bel`e1,
                                                e2:=org,
                                                e3:=skolem
                                        #)))
                        )
                ENDIF
ENDCASES

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

correctness_AM_JC_base: LEMMA
(FORALL (
        sw: t_States_w,
        lw: t_Labels_w
):
        joinCoalition_w?(lw) AND newAuth(lw) = empty_seq
        IMPLIES
        sigma(NEXT_w(sw, lw)) = TERMINAL_y(sigma(sw), alpha(sigma(sw), lw))
)


correctness_AM_JC_step: LEMMA
(FORALL (
        sw: t_States_w,
        lw2: t_Labels_w,
        lw1: t_Labels_w,
        na: t_TripleString
):
        joinCoalition_w?(lw1) AND
        joinCoalition_w?(lw2) AND
        org(lw1) = org(lw2) AND
        newAuth(lw1) = allButLast(newAuth(lw2)) AND
        newAuth(lw2) = concat(newAuth(lw1), sequence(na)) AND

        sigma (NEXT_w(sw, lw1)) = TERMINAL_y( sigma(sw), alpha(sigma(sw), lw1))
        IMPLIES
        sigma(NEXT_w(sw, lw2)) = TERMINAL_y( sigma(sw), alpha(sigma(sw), lw2))
)

correctness_AM: LEMMA
(FORALL (
        sw: t_States_w,
        lw: t_Labels_w
):
        sigma ( NEXT_w(sw, lw)) = TERMINAL_y( sigma(sw), alfa(sy, lw))
)

correctness_QM: LEMMA
(FORALL (
        sw: t_States_w,
        qw: t_Queries_w
):
        THEORY_w(sw)(qw) IFF pi(qw, THEORY_y(sigma(sw)))
)

END coalition_AMa
```

CITED LITERATURE

1. Ammann, P., Lipton, R. J., and Sandhu, R. S.: The expressive power of multi-parent creation in monotonic access control models. Journal of Computer Security, 4(2/3):149–166, 1996.
2. Li, N., Mitchell, J. C., and Winsborough, W. H.: Beyond proof-of-compliance: security analysis in trust management. Journal of the ACM, 52(3):474–514, 2005.
3. Sandhu, R.: Expressive power of the schematic protection model. Journal of Computer Security, 1(1):59–98, 1992.
4. Bertino, E., Catania, B., Ferrari, E., and Perlasca, P.: A logical framework for reasoning about access control models. ACM Transactions on Information and System Security, 6(1):71–127, 2003.
5. Harrison, M. A., Ruzzo, W. L., and Ullman, J. D.: Protection in operating systems. Communications of the ACM, 19(8):461–471, 1976.
6. Lipton, R. J. and Snyder, L.: A linear time algorithm for deciding subject security. Journal of the ACM, 24(3):455–464, 1977.
7. Chander, A., Mitchell, J. C., and Dean, D.: A state-transition model of trust management and access control. In CSFW, pages 27–43, 2001.
8. Tripunitara, M. V. and Li, N.: A theory for comparing the expressive power of access control models. Journal of Computer Security, 15(2):231–272, 2007.
9. Osborne, S., Sandhu, R., and Munawer, Q.: Configuring role-based access control to enforce mandatory and discretionary access control policies. ACM Transactions on Information and System Security, 3(2):85–106, May 2000.
10. Sandhu, R. and Ganta, S.: On testing for absence of rights in access control models. In Proceedings of the Sixth Computer Security Foundations Workshop (CSFW), pages 109–118, June 1993.
11. Hinrichs, T. L., Martinoia, D., Garrison III, W. C., Lee, A. J., Panebianco, A., and Zuck, L.: Application-sensitive access control evaluation using parameterized expressiveness. `http://www.cs.uic.edu/~hinrichs/papers/hinrichs2013towards-full.pdf`, 2013.
12. U.S. Air Force Scientific Advisory Board: Networking to enable coalition operations. Technical report, MITRE Corporation, 2004.
13. Horizontal integration: Broader access models for realizing information dominance. Technical Report JSR-04-13, MITRE Corporation JASON Program Office, December 2004.

14. Floyd, R. W.: Assigning meanings to programs. In Proceedings of the American Mathematical Society Symposia on Applied Mathematics, volume 19, pages 19–31, 1967.

VITA

NAME:	Diego Martinoia
EDUCATION:	B.S., Engineering of Computing Systems, Politecnico di Milano, Milano, Italy, 2011
	M.S., Computer Science, University of Illinois at Chicago, Chicago, Illinois, 2013
TEACHING:	December 2011 - Present (with interruptions): High school laboratory teacher of Computer Science, Computer Systems, Math and Statistics. Various locations.
PUBLICATIONS:	Martinoia, D.: "Questioning Hu's Invariants - Bad or Good Enough?" in VISAPP Volume 1: 311-316, 2012.