RZ 3276 (# 93322) 09/25/00
Computer Science 58 pages

Research Report

An Enhancement and Implementation for the MIERA Scheme

Simon Kramer*

IBM Research
Zurich Research Laboratory
8803 Rüschlikon
Switzerland

*Simon Kramer is at the Swiss Federal Institute of Technology (EPFL), 1015 Lausanne, Switzerland. This work was carried while he was a summer student at the IBM Zurich Research Laboratory in 1999.

Research
Almaden · Austin · Beijing · Delhi · Haifa · T.J. Watson · Tokyo · Zurich

An Enhancement and Implementation for the MIERA Scheme

Simon Kramer*

IBM Research, Zurich Research Laboratory, 8803 Rüschlikon, Switzerland
E-mail: simon.kramer@acm.org

Abstract

This paper describes an enhancement and implementation of an existing inter-enterprise role-based authorisation scheme for electronic transactions in the context of the Internet. For the enhancement, an abstract model and proof of its essential properties are given. The implementation is documented by text and class diagrams. Finally, some screenshots of a running demonstrator application are provided to illustrate the working of the scheme.

*Simon Kramer is at the Swiss Federal Institute of Technology (EPFL), 1015 Lausanne, Switzerland. This work was carried while he was a summer student at the IBM Zurich Research Laboratory in 1999.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The objective of this report is to describe an enhancement of the original MIERA (pronounced meer-rah) scheme as suggested in [1] and to give a detailed documentation of the current implementation.

This introduction describes the domain where the MIERA scheme could be applied to, defines the concepts and terms used in this report and gives a functional overview of the scheme. We conclude with suggesting the enhancement for the original scheme (called MIERA+). In Chapter 2 we show that the enhancement actually is one by proving two essential properties. Chapter 3 describes an implementation of the core functionalities of MIERA and the implementation of an interactive demonstrator for them.

The implementations were carried out during a Summer Stage at the IBM Research Laboratory, Zurich, Switzerland in 1999. The report has been written as a semester work at the Federal Institute of Technology in Lausanne (EPFL), Switzerland.

I would like to thank the creators of MIERA, namely Dr. Luke O'Connor and Dr. Heiko Ludwig for reviewing this report, Dr. Didier Buchs of the Software Engineering Laboratory at the EPFL for his assistance during the semester work and Prof. Jacques Zahnd from the EPFL for allowing me to use his LaTeX macros for the proofs in Section 2.3.

1.1 Application Domain

We consider electronic transactions in the context of the Internet. The transactions are typically interchanged between a service providing company, called PROVIDER, and a service requesting company, called CLIENT. MIERA (Method for Inter-Enterprise Role-based Authorisation) gives a possible solution to the transaction authorisation problem TAP defined in [1] as follows:

> How can a transaction requestor ... convince a transaction verifier ... that the requestor is authorized to request a specific transaction T?

The central idea of MIERA is, according to [1], that

> ... users[1] authorise transactions by signing some representation of the transaction, and conversely, verifiers determine that a transaction is authorized by verifying signatures.

and that

> ... the only required semantics is that if a given signature on the transaction verifies, then the requestor is defined as being authorised to make the transaction.

1.2 Concepts and Terminology

We expect the reader to be familiar with the basic concepts of public key cryptography and with the concepts of hash functions and certificates. It would also be convenient for the reader to be familiar with the content of [1]. Specific concepts and terms referred to in this report then, are defined in Tables 1.1 and 1.2.

1.3 A Functional Overview

1.3.1 The Transaction Form

There are basically only two operations to be performed on the transaction form. They correspond to the Items 4 and 5 in Table 1.2. Filling out does not need any comment. The collecting of the signatures does not belong to the core functionalities of MIERA and can either be done manually or by a process like a work flow system.

1.3.2 The Authorisation Tree

1.3.2.1 The Classical Scheme

The MIERA scheme as it is conceived in [1] transmits directly the hashes of the selected role names and the hashes of the remaining permission sets. However, the role name hashes are not sent unprotected but rather encapsulated in anonymous role certificates. This is because a verifier does not only have to check the signatures on the transaction form using these anonymous role certificates but he also have to calculate the root hash based on role name hashes extracted from these anonymous role certificates.

Once the verifier has extracted the above role name hashes, he will concatenate them in an order that has to be indicated in the transaction and calculate the hash of the resulting string. He then concatenates the obtained hash value with the permission set hash values in an order that has also to be indicated in the transaction and calculates the hash of the resulting string, thus obtaining the root hash.

If the role name hashes really do form a permission set then the root hash and the reference hash will be equal and the transaction is defined to be authorised. In the other case the transaction requestor either has forgotten to collect a

[1] Users can be machines or humans.

Concept or Term	**Definition**	**Example(s)**
Role of an employee	Function assumed by an employee of a certain company. The employee will be called the owner of that role.	
Absolute role		*Manager*, *Summer Student*
Relative role	<*absolute role*> + <*relational extension*>	*Manager of E-Business Solutions*, *Summer Student of E-Business Solutions*
Unique relative role	<*relative role*> + <*uniqueness extension*>, where <*uniqueness extension*> can be any symbol that is used only once for constructing this unique relative role name.	*Summer Student of E-Business Solutions:1*
Transaction type		*travel request*, *computer purchase*
Transaction form for a transaction of type T	Electronic form acting as a container for the information that is needed to process the transaction.	HTML-form
Permission set for a transaction of type T	Set of unique relative roles whose owners can authorise only jointly the transaction by signing the corresponding transaction form.	{ *Manager of E-Business Solutions, Summer Student of E-Business Solutions:1* }
Hash scheme	Set of rules defining how to create a hash value from a complex data structure.	If for a tree data structure *tree* the rules are defined as: • $rule1$ "Hash leaves as atomic values." • $rule2$ "Hash nodes by hashing the concatenation of the hashes of its descendants." then the resulting hash scheme is: $\{rule1, rule2\}$
Hash tree	Any tree data structure for which there exsists a corresponding hash scheme.	$(tree, hashScheme(tree))$

Table 1.1: Concepts and Terms 1

Concept or Term	**Definition**
Authorisation tree for transactions of type T	Tree that expresses the authorisation structure of transactions of type T. The tree is such that: the root node bears the label T, from the root node are pending nodes denoting the permission sets for T and bearing each one an identifier for the corresponding permission set.
Reference root hash	Hash created from an authorisation tree according to the correspondig hash scheme. The hash is signed by the local certificate authority and stored in a public data base $rhsPDB$.
Anonymous role certificate	Certificate that instead of containing an owner's identity contains the hash of a role name. The anonymous role certificate is kept in a public data base $ARCsPDB$.
Transaction request for atransaction of type T	Procedure that given the following definitions: • tf the blank transaction form • at the corresponding authorisation tree and • rh the certified reference hash consists in: 1. choosing a permission set ps from at 2. obscuring in at any information that is not related to ps. This gives us $obscure(at)$ 3. retrieving rh from $rhPDB$ 4. filling out tf 5. collecting the signatures on tf stipulated by ps 6. retrieving the signers' anonymous role certificates $ARCs$ from $ARCsPDB$ 7. dispatching the transaction t to PROVIDER where $$t = (obscure(at), rh, tf, ARCs)$$
Requestor	Employee of CLIENT making a transaction request.
Transaction verification	Procedure that consists in verifying: • the integrity and authenticity of: rh, tf and the $ARCs$ • that $vh = rh$ where $vh = hash(obscure(at))$
Verifier	Employee of PROVIDER that carries out the transaction verification for a certain incoming transaction.

Table 1.2: Concepts and Terms 2

signature on the transaction form or has consciously tried to cheat by leaving out on a (necessary) authoriser.

In the classical scheme, the number of permission sets is disclosed. Can we do better and also conceal the number of permission sets?

1.3.2.2 MIERA+

We actually can, at least on average. The way the authorisation information is being transmitted already suggests a solution. We basically transmit two sets: the role name hashes and the hashes of the remaining permission sets. It would be nice if we could hash up the set of permission set hashes and transmit its value instead of leaving this hashing to the verifier and thus disclosing the number of permission sets. The only problem is that our[2] hash function returns a different value for the concatenation of a string a and a string b than it does for the concatenation of b and a. This is why we have to respect order *within* a permission set and *between* permission sets when concatenating the hashes of their respective elements (see Section 1.3.2.1).

Thus, normally the verifier will have to insert the hash of the concatenation of the role name hashes at some position in the sequence of permission set hashes to respect the stipulated order between permission set hashes. Therefore, we normally cannot perform the hashing of the permission set hashes for the verifier and we are forced to disclose the number of permission sets. But let's do it when we can and try to maximise the number of times we can do it. The case where we can perform hashing for the verifier is when the selected permission set is the leftmost (or rightmost) in the authorisation tree. In this case, insertion, i.e. concatenation of three substrings, simplifies to concatenation of two substrings[3] and we *can* perform hashing of the permission set hashes for the verifier.

Yet, what if the requestor does not select the leftmost permission set in the authorisation tree but for example selects the second leftmost? Well, we nevertheless stick to our idea and perform the hashing on the rest of the permission set hashes to the right of it, and so on. The worst case of our method is obviously the one where the requestor selects the rightmost permission set in the authorisation tree, which means that we cannot perform any hashing on the concatenation of the permission set hashes at all.

If we want to implement our method, we need an appropriate data structure that relates the various hashes in the way the method works. A natural solution is to take a binary tree forming a main axis with permission sets pending as direct left descendent subtrees. This tree data structure can always be flattened to a linear list of the form $(PS_1, (PS_2, ...(PS_{n-1}, PS_n)))$ if $PS_1, ..., PS_n$ denote the permission sets in the authorisation tree. This linear list structure allows us also to maximise the number of times we can conceal the maximum amount of information about the number of permission sets. We just declare the permission set most frequently used by requestors as $PS1$. Such a permission set fortunately exists because in a real world context a transaction of a certain type is most of the times authorised by one and the same set of authorisers (permission set).

[2] in fact any hash function if it is to be of any use.

[3] the first substring being the one that results from the concatenation of the role name hashes and the second being the one that results from the concatenation of the permission set hashes.

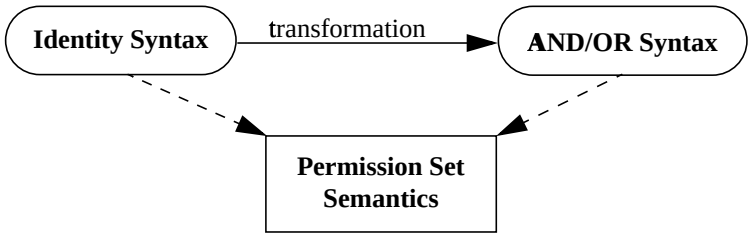

Figure 1.1: Syntax Transformation

Our method has lead us to modify the tree topology of the original authorisation tree, which actually encodes the semantics of the corresponding transaction. Tree topology therefore corresponds to the syntax of some imaginary grammar[4] that generates our trees. From our binary tree we can always reconstruct the original authorisation tree because the flattened list representation of a binary tree $(PS_1, (PS_2, ...(PS_{n-1}, PS_n)))$ can always be transformed into a representation of the form $(PS_1, PS_2, ..., PS_{n-1}, PS_n)))$, which is isomorph to the original tree. In fact, our method is a syntax transformation that preserves the meaning of the encoded concept (the concept of a series of permission sets). Any other syntax transformation that does not preserve this meaning would obviously be unacceptable and useless.

The classical as well as the enhanced scheme are essentially based on a static authorisation information structure, i.e. a tree that once defined for a certain transaction type does not change anymore[5]. Both methods are therefore intrinsically prone to an attack that considers a *series* of authorisation trees of one and the same transaction type. In this case it is possible that an intruder[6] ends up knowing the whole authorisation tree structure of a certain transaction type T. More precisely, we run this risk iff for a series of transactions of type T there exists a subset of transactions such that each transaction has been authorised by a different permission set and that all these permission sets essentially form the authorisation tree for T.

1.3.3 Putting it to work

1.3.3.1 The Normal Case

Figure 1.2 gives a global view of the authorisation process of an example transaction. The transaction is of type T with an authorisation tree GAT having permission sets $PS1 = \{R1, R2, R3\}$, $PS2 = \{R3, R5\}$ and $PS3 = \{R6\}$. The requestor has selected the roles $R1$, $R2$ and $R3$ for transaction authorisation.

GAT is then transformed into the binary tree PAT as suggested by the enhancement. PAT focuses on the logical structure of the authorisation changing

[4]For the original authorisation tree at least, the corresponding grammar is not so imaginary because the transaction administrator has to respect certain rules when defining the authorisation tree for each transaction type. These rules can be viewed as the productions of this grammar.

[5]at least as long as the authorisation structure it encodes does not change.

[6]The verifier itself is of course the most likely 'intruder.'

the permission set labelling of GAT, which focuses on the identities of the permission sets, into a labelling reflecting the logical relation between roles (see Figure 1.1). The reference hash rh has previously been calculated by the transaction authority and certified by the local certificate authority.

The requestor proceeds by hashing the role names, and the permission sets that are not directly used in this current authorisation. The resulting tree (TAT) is transformed into an intermediate representation IR, which is sent together with the anonymous role certificates, the reference hash and the signed transaction form to PROVIDER. The intermediate representation comprises an explicit declaration of the hashes that correspond to the selected role names (ps_h).

The verifier will then check that the hashes in ps_h match with the role name hashes extracted from the anonymous role certificates ($ps_h \in ARCs$) and reconstruct the tree based on IR and according to the labelling scheme mentioned in Figure 1.2. From the resulting tree VT, he then calculates the hash vh and compares it to rh. If all checks are valid and the signatures on the transaction form verify then the transaction is accepted as authorised, otherwise it is rejected.

Hashing is done according to so called *hash schemes*. The hash scheme of PAT must essentially be the same as the hash scheme of GAT because hashing depends on the semantics of the authorisation tree. However, the hash scheme of PAT is somewhat more secure than the hash scheme of GAT because it hashes OR-nodes two-by-two and not all together [7]. Both hash schemes would create exactly the same values if we did not do this two-by-two hashing and so the syntax transformation would be a homomorphism from GAT into PAT. In fact, it would even be an isomorphism because there exists $norm^{-1}$.

1.3.3.2 Cheating

What if the requestor does not provide sufficiently many signatures on the transaction form? Figure 1.3 shows the case where the requestor has not provided the signature of role $R2$, a case which the labelling scheme detects and which in turn causes the hashing to produce a value vh that is different to rh. The transaction is therefore rejected.

The requestor could also construct an authorisation tree where $R2$ is absent. So, respectively to this tree, the requestor would have provided 'all' role signatures. Yet again, hashing would detect cheating due to the resulting hash value, which is ensured to be different to the hash value of the correct authorisation tree.

For a more detailed treatment of cheating, the reader is referred to Section 2.2.4. There we analyse different ways of cheating and examine which counter measures are sufficient to tackle each one of them.

[7] We can look at the hash scheme of PAT as a *refinement* of the hash scheme of GAT.

Figure 1.2: The Normal Case

Figure 1.3: A Case of Cheating

Chapter 2

An abstract model for MIERA+

2.1 Introduction

The goal of this chapter is to give a model for MIERA+ that is formal and abstract (in the sense that it is machine independent). We describe the approach we have followed to build the model, and the language we have used to express it. The main part of this chapter is devoted to the model itself and the proof of its main properties.

2.1.1 Approach

We follow a so called *constructive*, as opposed to *deductive*, specification approach. Table 2.1 gives a comparative view of the characteristics of constructive and deductive system specification.

In a deductive approach we define the system we have in mind by stating its properties. We then refine these properties into code using a *refinement calculus*. The obtained code represents a *concrete* model of our system. Correctness is directly defined by the giving of the system properties and is maintained in the

	Approach	
Characteristics	Deductive	Constructive
Specification syntax	a conjunction of properties	an algebra: a support and a set of operations
System properties	explicit	implicit
System	implicitly defined (by intension)	explicitly defined (by extension)
Refinement into code	complex	straight forward

Table 2.1: Constructive and Deductive System Specification: Characteristics

code by the refinement process.

In a constructive approach we start by defining an *abstract* model of our system. The model is abstract in the sense that it is mathematical and machine independent. We then proceed by formulating the properties that our system is supposed to have. Finally, we must prove that these properties are in fact theorems of our mathematical model and by doing so we establish its correctness.

2.1.2 Scope

Our situation is such that the code already exists and we therefore should reason directly on it in order to establish correctness. We will not do this because the implementation may change and also because reasoning on code takes generally a lot of time.

Thus it seems natural to follow a constructive approach and define an abstract model of the mechanism, i.e. the enhanced authorisation process, we want to prove.

2.1.3 Formalism: CO-OPN/2 [1]

CO-OPN/2 is based on two underlying formalisms, namely order-sorted partial algebras and Petri nets. The combination of both is called *algebraic net*. In this type of net, places and tokens become algebraic values. Thus, order-sorted partial algebras are used to describe data structures, more precisely, data structures of a system specification, while Petri nets are used to model the behavioural and concurrent aspects of the system. To the concept of algebraic net CO-OPN/2 adds object-orientation to remedy the lack of structuration present in both order-sorted partial algebras and Petri nets. A system is considered as a collection of autonomous objects grouped together in a class.

A CO-OPN/2 specification consists of two kinds of modules: ADT (Abstract Data Type) modules and class modules[2]. ADT modules describe the various data structures of the system specification, whereas a class module describes an algebraic net representing a class of objects that share a common internal structure. Both kinds of modules have a header and three other, optional sections, namely `Inherit`, `Interface` and `Body`. The header always states the name of the module and its type, i.e. `ADT` or `class`, and nothing else. The section `Inherit` groups together, as its name indicates, all information related to inheritance. The section `Interface` defines what and how information can be accessed by other classes of the system and the section `Body` contains private data, i.e. data that cannot be accessed from outside the module.

In an ADT module, the section `Interface` can contain the subsections `Use`, `Sorts`, `Generators` and `Operations`, whereas the section `Body` can contain the subsections `Operations` and `Axioms`.

If the ADT module uses any other element defined in the interface of another module, then the sort of this element must be stated in the subsection `Use`. The subsection `Sorts` enumerates all the names of the types that the ADT mentions in its proper definition. `Generators` enumerates all the declarations of the operations that can generate values of the new type defined by the ADT. The arity and the position of the name of the operation relative to its

[1] The present description is mainly drawn from [2].

[2] We do not need class modules for our problem description.

parameters (pre-, in- and postfix) are implicit in the notation. For example, if the string *emptygat* _ : *transactiontype* → *gat* is mentioned in the subsection `Generators`, then this means first, that the ADT defines a prefix operation with name `emptygat` and second, that this operation takes a single parameter of type `TransactionType` generating a value of type `GAT`. The subsection `Operations` enumerates the declarations of any auxiliary operations. The subsection `Axioms` represents the definition part of the operations in the ADT. Operations are defined by the giving of their properties, which take the form of equations.

2.2 Problem Description

2.2.1 The Abstract Model

The abstract model consists of the CO-OPN/2 package *MIERA*, which contains two sets of ADT modules that form together the authorisation process specification. The first set contains the ADT modules for the general authorisation tree and its constituting components, the permission set nodes and the roles (see Tables from 2.2 to 2.4). The second set of ADT modules contains the specification of the verification tree (see Tables 2.7 and 2.8).

In the authorisation process the general authorisation tree undergoes various transformations that subsequently take it through the state of a publishable authorisation tree, a transmittable authorisation tree, an intermediate representation and finally a verification tree.

For the sake of simplicity, the multiple transformations have been condensed into a single transformation, called `norm`. It takes the general authorisation tree, called *source tree*, as input and generates the verification tree, called *target tree*.

Roles have exactly two states: either selected or not selected. If a certain role is selected, then this means that the requestor has correctly enclosed the corresponding anonymous role certificate in the transaction. The contrary holds in the opposite case. This selection mechanism models the correspondance check between the hashes in the tree container and the hashes in the form container.

A *valid* general authorisation tree is one where there exists at least one permission set such that all roles are marked as selected. This definition of validity also provides for the unlikely but nevertheless possible case where the requestor sends to the verifier a form that bears multiple sets of signatures each one corresponding to a different permission set.

```
ADT GeneralAuthorizationTree;

Interface

    Use
        Booleans;
        VerificationTree;
        TransactionType;
        PermissionSetNode;

    Sort
        gat;

    Generators
        emptygat _ : transactiontype -> gat;
        addpsn _ _ : psn gat -> gat;

    Operations
        norm _ : gat -> vt;
        isvalid _ : gat -> boolean;

Body

    Operations
        isempty _ : gat -> boolean;
        isordered _ : gat -> boolean;
        _ existsin _ : psn gat -> boolean;
        _ > _ : psn gat -> boolean;

    Axioms
        addpsn1 : (isempty node) = true | (node existsin tree) = true
                    => addpsn node tree = tree;
        addpsn2 : (node existsin tree) = false & (isempty node) = false
                    => addpsn node tree = addpsn node tree;
        isordered1 : isordered (emptygat t) = true;
        isordered2 : isordered (addpsn node tree) = (node > tree) and (isordered tree);
        isempty1 : isempty (emptygat t) = true;
        isempty2 : isempty (addpsn node tree) = false;
        isvalid1 : isvalid (emptygat t) = false;
        isvalid2 : (isempty tree) = true => isvalid (addpsn node tree) = isvalid node;
        isvalid3 : (isempty tree) = false
                     => isvalid (addpsn node tree) = (isvalid node) or (isvalid tree);
        norm1 : norm (emptygat t) = emptyvt;
        norm2 : (isempty tree) = true => norm (addpsn node tree) = norm node;
        norm3 : (isempty tree) = false
                  => norm (addpsn node tree) = (OR addsub (norm node) (norm tree));
        existsin1 : node existsin (emptygat t) = false;
        existsin2 : no1 existsin (addpsn no2 tree)
                      = (no1 = no2) or (no1 existsin tree);
        greater1 : node > emptygat t = true;
        greater2 : no1 > (addpsn no2 tree) = (no1 > no2) and (no1 > tree);

    Theorems
        T1 : isvalid tree = isvalid (norm tree);
        T2 : (norm g = v) = false => isvalid v = false;

    Where
        t : transactiontype;
        tree : gat;
        node : psn;
        no1 : psn;
        no2 : psn;
        g : gat;
        v : vt;

End GeneralAuthorizationTree;
```

Table 2.2: Source Tree Modules: `GeneralAuthorisationTree`

```
ADT PermissionSetNode;

Interface

    Use
        Role;
        Label;
        Booleans;
        VerificationTree;
        PermissionSetName;

    Sort
        psn;

    Generators
        emptypsn _ : permissionsetname -> psn;
        addroleleaf _ _ : role psn -> psn;

    Operations
        norm _ : psn -> vt;
        isvalid _ : psn -> boolean;
        isempty _ : psn -> boolean;
        _ existsin _ : role psn -> boolean;
        isordered _ : psn -> boolean;
        _ > _ : psn psn -> boolean;
        _ = _ : psn psn -> boolean;

Body

    Operations
        allselected _ : psn -> boolean;
        name _ : psn -> permissionsetname;
        _ > _ : role psn -> boolean;

    Axioms
        addroleleaf1 : (r existsin no = true) => addroleleaf r no = no;
        addroleleaf2 : (r existsin no = false) => addroleleaf r no = addroleleaf r no;
        isvalid1 : isvalid no = allselected no;
        norm1 : norm (emptypsn nna) = emptyvt;
        norm2 : (isempty no) = true => norm (addroleleaf r no) = leaf r;
        norm3 : (isempty no) = false => norm (addroleleaf r no)
                  = (AND addsub (leaf r) (norm no));
        isempty1 : isempty (emptypsn nna) = true;
        isempty2 : isempty (addroleleaf r no) = false;
        existsin1 : r existsin (emptypsn nna) = false;
        existsin2 : r1 existsin (addroleleaf r2 no) = (r1 = r2) or (r1 existsin no);
        isordered1 : isordered (emptypsn nna) = true;
        isordered2 : isordered (addroleleaf r no) = (r > no) and (isordered no);
        greater1 : no1 > no2 = (name no1) > (name no2);
        equal1 : (no1 = no2) = ((name no1) = (name no2));
        allselected1 : allselected (emptypsn nna) = false;
        allselected2 : isempty no = true
                          => allselected (addroleleaf r no) = isselected r;
        allselected3 : isempty no = false => allselected (addroleleaf r no)
                          = (isselected r) and (allselected no);
        name1 : name (emptypsn nna) = nna;
        name2 : name (addroleleaf r no) = name no;
        greater1 : r > (emptypsn nna) = true;
        greater2 : r1 > (addroleleaf r2 no) = (r1 > r2) and (r1 > no);

    Where
        nna : permissionsetname;
        nna1 : permissionsetname;
        nna2 : permissionsetname;
        r : role;
        r1 : role;
        r2 : role;
        no : psn;
        no1 : psn;
        no2 : psn;

End PermissionSetNode;
```

Table 2.3: Source Tree Modules: `PermissionSetNode`

```
ADT Role;

Interface

    Use
        Booleans;
        String;

    Sort
        role;

    Generator
        newrole _ _ : string boolean -> role;

    Operations
        isselected _ : role -> boolean;
        _ > _ : role role -> boolean;
        _ = _ : role role -> boolean;

Body

    Axioms
        isselected1 : isselected (newrole na1 true) = true;
        isselected2 : isselected (newrole na1 false) = false;
        greater1 : (newrole na1 v1) > (newrole na2 v2) = na1 > na2;
        equals1 : ((newrole na1 v1) = (newrole na2 v2)) = (na1 = na2);

    Where
        na1, na2 : string;
        v1, v2 : boolean;

End Role;
```

Table 2.4: Source Tree Modules: `Role`

```
ADT TransactionType;

Interface

    Sort
        transactiontype;

End TransactionType;
```

Table 2.5: Source Tree Modules: `TransactionType`

```
ADT PermissionSetName As String;

Rename

    string -> permissionsetname;

End PermissionSetName;
```

Table 2.6: Source Tree Modules: `PermissionSetName`

Table 2.7: Target Tree Modules: `VerificationTree`

```
ADT VerificationTree;

Interface

    Use
        Role;
        Label;
        Booleans;

    Sort
        vt;

    Generators
        emptyvt : -> vt;
        leaf _ : role -> vt;
        _ addsub _ _ : label vt vt -> vt;

    Operations
        isvalid _ : vt -> boolean;
        _ = _ : vt vt -> boolean;
        relabel _ : vt -> vt;

Body

    Operations
        topok _ : vt -> boolean;
        topokps _ : vt -> boolean;
        evps _ : vt -> boolean;
        isleaf _ : vt -> boolean;
        isvps _ : vt -> boolean;
        isnode _ : vt -> boolean;
        isselectedrole _ : vt -> boolean;

    Axioms
        isvalid1 : isvalid tree = (topok tree) and (evps tree);
        topok1 : topok emptyvt = true;
        topok2 : topok (leaf r) = true;
        topok3 : topok (AND addsub no1 no2) = topokps (AND addsub no1 no2);
        topok4 : topok (OR addsub no1 no2)
                   = (topokps no1) and ((topok no2) or (topokps no2));
        topokps1 : topokps emptyvt = true;
        topokps2 : topokps (leaf r) = true;
        topokps3 : topokps (l addsub no1 no2) = (isleaf no1) and (topokps no2);
        evps1 : evps emptyvt = false;
        evps2 : evps (leaf r) = isselectedrole (leaf r);
        evps3 : evps (AND addsub no1 no2) = isvps (AND addsub no1 no2);
        evps4 : evps (OR addsub no1 no2) = (isvps no1) or (evps no2);
        isselectedrole1 : isselectedrole emptyvt = false;
        isselectedrole2 : isselectedrole (leaf r) = isselected r;
        isselectedrole3 : isselectedrole (l addsub no1 no2) = false;
        isleaf1 : isleaf emptyvt = false;
        isleaf2 : isleaf (leaf r) = true;
        isleaf3 : isleaf (l addsub no1 no2) = false;
        isvps1 : isvps emptyvt = false;
        isvps2 : isvps (leaf r) = isselectedrole (leaf r);
        isvps3 : isvps (AND addsub no1 no2) = (isvps no1) and (isvps no2);
        isvps4 : isvps (OR addsub no1 no2) = false;
        isnode1 : isnode emptyvt = false;
        isnode2 : isnode (leaf r) = false;
        isnode3 : isnode (l addsub no1 no2) = true;
        equal1 : (no1 = emptyvt) = true => (emptyvt = no1) = true;
        equal2 : (no1 = emptyvt) = false => (emptyvt = no1) = false;
        equal3 : (no1 = emptyvt) = true => (no1 = emptyvt) = true;
        equal4 : (no1 = emptyvt) = false => (no1 = emptyvt) = false;
        equal5 : ((leaf r1) = (leaf r2)) = (r1 = r2);
        equal6 : (isleaf no1) = false => ((leaf r) = no1) = false;
        equal7 : (isleaf no1) = false => (no1 = (leaf r)) = false;
        equal8 : (l1 addsub no1 no2) = (l2 addsub no3 no4)
                   = (l1 = l2) and ((no1 = no3) and (no2 = no4));
        equal9 : (isnode no3) = false => ((l addsub no1 no2) = no3) = false;
        equal10 : (isnode no3) = false => (no3 = (l addsub no1 no2)) = false;
```

Table 2.7: Target Tree Modules: `VerificationTree`

```
        relabel1 : relabel emptyvt = emptyvt;
        relabel2 : relabel (leaf r) = (leaf r);
        relabel3 : (isvalid tree) = true => relabel tree = tree;
        relabel4 : (isvalid (l addsub no1 no2)) = false
                     => relabel (l addsub no1 no2)
                          = (OR addsub (relabel no1) (relabel no2));

    Where
        r : role;
        r1 : role;
        r2 : role;
        tree : vt;
        no1 : vt;
        no2 : vt;
        no3 : vt;
        no4 : vt;
        l : label;
        l1 : label;
        l2 : label;

End VerificationTree;
```

```
ADT Label;

Interface

    Use
        Booleans;

    Sort
        label;

    Generators
        AND : -> label;
        OR : -> label;

    Operation
        _ = _ : label label -> boolean;

Body

    Axioms
        equals1 : (AND = AND) = true;
        equals2 : (AND = OR) = false;
        equals3 : (OR = AND) = false;
        equals4 : (OR = OR) = true;

End Label;
```

Table 2.8: Target Tree Modules: `Label`

2.2.2 Axioms

In the ADT module `GeneralAuthorisationTree`:

- The conditions in `addpsn1` ensure that a source tree never contains an empty permission set node, respectively that permission sets are unique within a given source tree.

- `isordered1` and `isordered2` establish the order relation over the set of permission sets, which is necessary for applying a hash scheme to an authorisation tree (see Section 1.3.2.2).
- `isvalid1`, `isvalid2` and `isvalid3` implement the concept of validity as described in Section 2.2.1.
- `norm1`, `norm2` and `norm3` implement the transformation explained in Section 2.2.1.

Axioms with analogous purposes and names are defined in the ADT module `PermissionSetNode`.

The main axiom in the ADT module `VerificationTree` is `isvalid1`. It decomposes the validity check of a verification tree into a topology check (`topok`) and a check for valid permission sets (`evps`). Topology checking is done globally on the whole tree (`topok`) and locally on subtrees that should correspond to permission sets (`topokps`). The same is true for permission set checking, which is performed on the whole tree (`evps`) as well as on supposed permission sets (`isvps`).

Relabelling of invalid verification trees, which is needed in order to make use of a hash scheme, is specified by `relabel1`, `relabel2`, `relabel3` and `relabel4`, which relabel an invalid permission set by simply giving it an 'OR'-label.

The presentation (called *présentation gracieuse* in French) of axioms is such that the domains they range over are disjoint. A consequence thereof is that our specification is *sound*, meaning that no contradiction can be derived from. If our specification was unsound, then any stated theorem would be trivially valid, which is definitely not what we want.

2.2.3 Relevant Properties

The transformation must conserve validity, i.e. a valid source tree must map onto a valid target tree and vice versa. The same must hold for invalid trees. This requirement is formulated as follows:

Requirement 1 (Conservation of Validity)

$$\forall(tree \in gat) \bullet isvalid(tree) = isvalid(norm(tree))$$

Target trees have, by construction, a well-defined topology. They are *well-formed*. Yet, not all trees that are due for verification may actually be target trees, i.e. trees that result from the application of `norm` to some source tree. Such trees are qualified as *bad trees* due to their inappropriate topology. Bad trees obviously are invalid. See Figure 2.1 for an illustration of this requirement.

Requirement 2 (Detection of Bad Trees)

$$\forall(g \in gat)\forall(v \in vt) \bullet (norm(g) = v) = false \Rightarrow isvalid(v) = false$$

Requirement 2 separates the class of verification trees into two disjoint subclasses, i.e. the class of well-formed trees and the class of bad trees. The separation criteria hereby is conformance to the tree topology as described in Section 1.3.2.2 and illustrated in Figure 1.2.

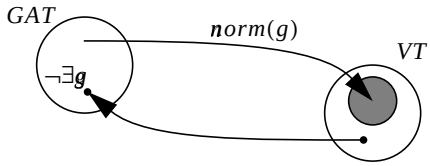

Figure 2.1: Requirement 2

Requirement 1 takes separation a step further by separating the class of well-formed verification trees, i.e. the class of target trees, into the class of valid target trees and the class of invalid target trees. A valid target tree is a target tree with valid topology and valid labelling whereas an invalid target tree is a target tree with valid topology but invalid labelling (see Table 2.9).

2.2.4 Pragmatics of Requirements 1 and 2

How do the different types of verification trees, i.e. bad trees, valid and invalid target trees relate to their potential creator the requestor, their transportation channel and their verifier?

A bad tree can either be due to a corruption occurring during transportation, or can be the result of a bad assembling process carried out by the requestor. The bad assembling can either be done unintentionally (by accident) or maliciously. 'Maliciously' means that the requestor has intentionally tried to construct a tree that favours the particular, incomplete role selection pattern chosen by him. An incomplete role selection pattern is a set of roles that the requestor wants to sign the transaction form and that does not contain at least one complete permission set.

An invalid target tree does not have a valid labelling, which means that the requestor has either forgotten to submit a necessary role signature, or yet that he tried to cheat by intentionally not submitting some role signature(s).

A valid target tree can either be the result of a rule-conform source tree transformation, or be the result of a malicious attempt similar to the one producing a bad tree, but this time the attempt is more successful because (the hypothetical) $norm^{-1}$ would transform it back into a (not necessarily the corresponding) valid source tree.

In order that the verifier can be sure that the valid target tree corresponds in fact to the correct source tree, he can calculate the *hash* of the target tree and verify that it is equal to the hash of the source tree.

Property 1 (Authorisation Tree Correctness) [3] Suppose that general authorisation trees are always ordered according to one and the same criteria[4].

Then if g is a source tree for transactions of type T, and v is some verification tree, then v is a *correct* authorisation tree for transactions of type T if and only if the properties $isvalid(v)$ and $hash(relabel(v)) = hash(g)$ hold.

[3] The evaluation of this property must be *lazy* because hashing is defined on valid target trees only.

[4] For example, according to the one specified by the axiom `isordered` in the ADT module `GeneralAuthorisationTree`. The hypothesis that prescribes that authorisation trees be ordered is necessary because no useful hash function can commute over tree structures.

	$isvalid(v)$		
	$true$	$false$	
Topology	$valid$	$valid$	$invalid$
Labelling	$valid$	$invalid$	$\oslash$
	$WELL-FORMED\_TREES = \{v \mid g \in GAT \Rightarrow v = norm(g)\}$		$BAD\_TREES = \{v \mid v \notin Im(norm)\}$
	$UNCONSTRAINED\_TREES$		

Table 2.9: What the function $isvalid : VT \rightarrow BOOLEAN$ expresses

2.3 Proofs

As a proof technique, structural induction over the ADTs `GeneralAutorizationTree`, `PermissionSetNode` and `VerificationTree` has been used. Universal quantifiers that range over variables that do not interfere with the proof have not been mentioned explicitly in the proof schemes. This is the case for variables of type `TransactionType` and `PermissionSetName`.

Proofs have been carried out using natural deduction. Their presentation follows a style, which is described in detail in [4].

The Proof of $T1$ and $T2$ have been split into several parts. They are shown in Figure 2.2 respectively in Figure 2.3. A full line between two proofs means that there is a relation of *decomposition*, whereas a pointed line between two proofs means that there is a relation of *use* between them.

Theorem $T1$

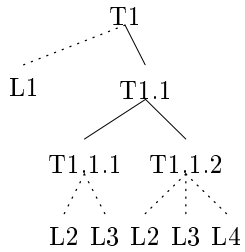

Figure 2.2: `T1`: Dependency Tree

	nil	
1°	$isvalid(emptygat(T)) = false$	gat:isvalid1
2°	$isvalid(norm(emptygat(T))) = isvalid(emptyvt)$	gat:norm1
	$= topok(emptyvt)\ and\ evps(emptyvt)$	vt:isvalid1
	$= true\ and\ false$	vt:topok1, vt:evps1
	$= false$	
3°	$isvalid(emptygat(T)) = isvalid(norm(emptygat(T)))$	1°, 2° — BC/A
4°	$isvalid(tree) = isvalid(norm(tree))$	IH/A
5°	$emptypsn(nm)\ existsin\ tree = true \mid emptypsn(nm)\ existsin\ tree = false$	
6°	$emptypsn(nm)\ existsin\ tree = true$	hyp
7°	$emptypsn(nm)\ existsin\ tree = false$	L1
8°	$\bot$	6°, 7°
9°	$isvalid(addpsn(emptypsn(nm), tree)) = isvalid(norm(addpsn(emptypsn(nm), tree)))$	8°, ex-falso
10°	$emptypsn(nm)\ existsin\ tree = false$	hyp
11°	$addpsn(emptypsn(nm), tree)) = tree$	6°, gat:addpsn1
12°	$isvalid(tree) = isvalid(norm(tree))$	4°
13°	$isvalid(addpsn(emptypsn(nm), tree)) = isvalid(norm(addpsn(emptypsn(nm), tree)))$	11°, 12°
14°	$isvalid(addpsn(emptypsn(nm), tree)) = isvalid(norm(addpsn(emptypsn(nm), tree)))$	5°, 9°, 13° — BC/B
15°	$isvalid(addpsn(ps, tree)) = isvalid(norm(addpsn(ps, tree)))$	IH/B
	⋮	Table 2.12
38°	$isvalid(addpsn(addroleleaf(rl, ps), tree)) = isvalid(norm(addpsn(addroleleaf(rl, ps), tree)))$	
39°	$isvalid(addpsn(ps, tree)) = isvalid(norm(addpsn(ps, tree)))$	
	$\Rightarrow isvalid(addpsn(addroleleaf(rl, ps), tree)) = isvalid(norm(addpsn(addroleleaf(rl, ps), tree)))$	15°, 38°
40°	$\forall(ps \in psn) \bullet isvalid(addpsn(ps, tree)) = isvalid(norm(addpsn(ps, tree)))$	14°, 39°, ind
41°	$isvalid(tree) = isvalid(norm, tree) \Rightarrow \forall(ps \in psn) \bullet isvalid(addpsn(ps(tree))) = isvalid(norm(addpsn(ps, tree)))$	4°, 40°
42°	$\forall(tree \in gat) \bullet isvalid(tree) = isvalid(norm(tree))$	3°, 41°, ind.

Table 2.10: Proof `T1`: top

	nil	
1°	$emptypsn(nm)\ existsin\ emptygat(T) = false$	gat:existsin1 — BC/A
2°	$emptypsn(nm)\ existsin\ tree = false$	IH/A
3°	$emptypsn(nm)\ existsin\ addpsn(emptypsn(nm), tree) = emptypsn(nm)\ existsin\ tree$	gat:addpsn1
	$= false$	2° — BC/B
4°	$emptypsn(nm)\ existsin\ addpsn(ps, tree) = false$	IH/B
5°	$emptypsn(nm)\ existsin\ addpsn(addroleleaf(rl, ps), tree)$	
	$= \{emptypsn(nm) = addroleleaf(rl, ps)\} or \{emptypsn(nm)\ existsin\ tree\}$	gat:existsin2
	$= false$	2°
6°	$emptypsn(nm)\ existsin\ addpsn(ps, tree) = false \Rightarrow emptypsn(nm)\ existsin\ addpsn(addroleleaf(rl, ps), tree) = false$	4°, 5°
11°	$\forall(ps \in psn) \bullet emptypsn(nm)\ existsin\ addpsn(ps, tree) = false$	3°, 6°, ind
12°	$emptypsn(nm)\ existsin\ tree = false \Rightarrow \forall(ps \in psn) \bullet emptypsn(nm)\ existsin\ addpsn(ps, tree) = false$	2°, 11°
13°	$\forall(tree \in gat) \bullet emptypsn(nm)\ existsin\ tree = false$	1°, 12°, ind.

Table 2.11: Proof T1: Lemma L1

	$\{IH/A\}$	
15°	$isvalid(addpsn(ps, tree)) = isvalid(norm(addpsn(ps, tree)))$	IH/B
16°	$rl\ existsin\ ps = true \mid rl\ existsin\ ps = false$	
17°	$rl\ existsin\ ps = true$	hyp
18°	$isvalid(addpsn(addroleleaf(rl, ps), tree)) = isvalid(addpsn(ps, tree))$	17°, psn:addroleleaf1
	$= isvalid(norm(addpsn(ps, tree)))$	15
	$= isvalid(norm(addpsn(addroleleaf(rl, ps), tree)))$	17°, psn:addroleleaf
19°	$rl\ existsin\ ps = false$	hyp
20°	$isempty(tree) = true \mid isempty(tree) = false$	
21°	$isempty(tree) = true$	hyp
22°	$norm(addpsn(addroleleaf(rl, ps), tree)) = norm(addroleleaf(rl, ps))$	21°, gat:norm2
23°	$isempty(ps) = true \mid isempty(ps) = false$	
24°	$isempty(ps) = true$	hyp
	⋮	Table 2.13
27°	$isvalid(addpsn(addroleleaf(rl, ps), tree)) = isvalid(norm(addpsn(addroleleaf(rl, ps), tree)))$	
28°	$isempty(ps) = false$	hyp
	⋮	Table 2.13
31°	$isvalid(addpsn(addroleleaf(rl, ps), tree)) = isvalid(norm(addpsn(addroleleaf(rl, ps), tree)))$	
32°	$isvalid(addpsn(addroleleaf(rl, ps), tree)) = isvalid(norm(addpsn(addroleleaf(rl, ps), tree)))$	23°, 27°, 31°
33°	$isempty(tree) = false$	hyp
	⋮	Table 2.16
36°	$isvalid(addpsn(addroleleaf(rl, ps), tree)) = isvalid(norm(addpsn(addroleleaf(rl, ps), tree)))$	
37°	$isvalid(addpsn(addroleleaf(rl, ps), tree)) = isvalid(norm(addpsn(addroleleaf(rl, ps), tree)))$	20°, 32°, 36°
38°	$isvalid(addpsn(addroleleaf(rl, ps), tree)) = isvalid(norm(addpsn(addroleleaf(rl, ps), tree)))$	16°, 18°, 37°

Table 2.12: Proof T1: Part T1.1

	$\{IH/A, rl\ existsin\ ps = false, isempty(tree) = true\}$	
23°	$isempty(ps) = true \mid isempty(ps) = false$	
24°	$isempty(ps) = true$	hyp
25°	$isvalid(norm(addpsn(addroleleaf(rl, ps), tree))) = isvalid(norm(addroleleaf(rl, ps)))$	21°, gat:norm1
	$= isvalid(leaf(rl))$	24°, psn:norm2
	$= topok(leaf(rl))\ and\ evps(leaf(rl))$	vt:isvalid1
	$= true\ and\ isselectedrole(leaf(rl))$	vt:topop2, vt:evps2
	$= isselected(rl)$	vt:isselectedrole2
26°	$isvalid(addpsn(addroleleaf(rl, ps), tree)) = isvalid(addroleleaf(rl, ps))$	21°, gat:isvalid2
	$= allselected(addroleleaf(rl, ps))$	psn:isvalid1
	$= isselected(rl)$	24°, psn:allselected2
27°	$isvalid(addpsn(addroleleaf(rl, ps), tree)) = isvalid(norm(addpsn(addroleleaf(rl, ps), tree)))$	
28°	$isempty(ps) = false$	hyp
29°	$isvalid(norm(addpsn(addroleleaf(rl, ps), tree)))$	
	$= isvalid(norm(addroleleaf(rl, ps)))$	21°, gat:norm2
	$= isvalid((AND\ addsub\ leaf(rl)\ norm(ps)))$	28°, psn:norm3
	$= topok((AND\ addsub\ leaf(rl)\ norm(ps)))\ and\ evps((AND\ addsub\ leaf(rl)\ norm(ps)))$	vt:isvalid1
	$= topokps((AND\ addsub\ leaf(rl)\ norm(ps)))\ and\ isvps((AND\ addsub\ leaf(rl)\ norm(ps)))$	vt:topok3, vt:evps3
	$= isleaf(leaf(rl))\ and\ topokps(norm(ps))\ and\ isvps(leaf(rl))\ and\ isvps(norm(ps))$	vt:topokps3, vt:isvps3
	$= true\ and\ true\ and\ isselectedrole(leaf(rl))\ and\ isvps(norm(ps))$	vt:isleaf2, L2, vt:isvps2
	$= isselected(rl)\ and\ isvps(norm(ps))$	vt:isselectedrole2
	$= isselected(rl)\ and\ allselected(ps)$	L3
30°	$isvalid(addpns(addroleleaf(rl, ps), tree)) = isvalid(addroleleaf(rl, ps))$	21°, gat:isvalid2
	$= allselected(addroleleaf(rl, ps))$	psn:isvalid1
	$= isselected(rl)\ and\ allselected(ps)$	28°, psn:allselected3
31°	$isvalid(addpsn(addroleleaf(rl, ps), tree)) = isvalid(norm(addpsn(addroleleaf(rl, ps), tree)))$	29°, 30°
32°	$isvalid(addpsn(addroleleaf(rl, ps), tree)) = isvalid(norm(addpsn(addroleleaf(rl, ps), tree)))$	23°, 27°, 13°

Table 2.13: Proof `T1`: Part `T1.1.1`

	nil	
1°	$topokps(norm(emptypsn(nm))) = topokps(emptyvt)$	gat:norm1
	$= true$	vt:topokps1 — BC
2°	$topokps(norm(ps)) = true$	IH
3°	$isempty(ps) = true \mid isempty(ps) = false$	
4°	$isempty(ps) = true$	hyp
5°	$topokps(norm(addroleleaf(rl, ps))) = topokps(leaf(rl))$	4°, psn:norm2
	$= true$	vt:topok2
6°	$isempty(ps) = false$	hyp
7°	$topokps(norm(addroleleaf(rl, ps))) = topokps((AND\ addsub\ leaf(rl)\ norm(ps)))$	6°, psn:norm3
	$= isleaf(leaf(rl))\ and\ topokps(norm(ps))$	vt:topokps3
	$= true$	vt:isleaf2, 2°
8°	$topokps(norm(addroleleaf(rl, ps))) = true$	3°, 5°, 7°
9°	$topokps(norm(ps)) = true \Rightarrow topokps(norm(addroleleaf(rl, ps))) = true$	2°, 8°
10°	$\forall(ps \in psn) \bullet topokps(norm(ps)) = true$	1°, 9°, ind.

Table 2.14: Proof T1: Lemma L2

	nil	
1°	$allselected(emptypsn(nm)) = false$	psn:allselected1
2°	$isvps(norm(emptypsn(nm))) = isvps(emptyvt)$	gat:norm1
	$= false$	vt:isvps1
3°	$allselected(emptypsn(nm)) = isvps(norm(emptypsn(nm)))$	1°, 2° — BC
4°	$allselected(ps) = isvps(norm(ps))$	IH
5°	$isempty(ps) = true \mid isempty(ps) = false$	
6°	$isempty(ps) = true$	hyp
7°	$allselected(addroleleaf(rl, ps)) = isselected(rl)$	6°, psn:allselected2
8°	$isvps(norm(addroleleaf(rl, ps))) = isvps(leaf(rl))$	6°, psn:norm2
	$= isselectedrole(leaf(rl))$	vt:isvps2
	$= isselected(rl)$	vt:isselectedrole2
9°	$allselected(addroleleaf(rl, ps)) = isvps(norm(addroleleaf(rl, ps)))$	7°, 8°
10°	$isempty(ps) = false$	hyp
11°	$allselected(addroleleaf(rl, ps)) = isselected(rl)\ and\ allselected(ps)$	10°, psn:allselected3
	$= isselected(rl)\ and\ isvps(norm(ps))$	4°
12°	$isvps(norm(addroleleaf(rl, ps))) = isvps((AND\ addsub\ leaf(rl)\ norm(ps)))$	10°, psn:norm3
	$= isvps(leaf(rl))\ and\ isvps(norm(ps))$	vt:isvps3
	$= isselectedrole(leaf(rl))\ and\ isvps(norm(ps))$	vt:isvps2
	$= isselected(rl)\ and\ isvps(norm(ps))$	vt:isselectedrole2
13°	$allselected(addroleleaf(rl, ps)) = isvps(norm(addroleleaf(rl, ps)))$	11°, 12°
10°	$allselected(addroleleaf(rl, ps)) = isvps(norm(addroleleaf(rl, ps)))$	5°, 9°, 13°
11°	$allselected(ps) = isvps(norm(ps)) \Rightarrow allselected(addroleleaf(rl, ps)) = isvps(norm(addroleleaf(rl, ps)))$	4°, 10°
12°	$\forall(ps \in psn) \bullet allselected(ps) = isvps(norm(ps))$	3°, 11°, ind.

Table 2.15: Proof T1: Lemma L3

$\{IH/A, IH/B, rl\ existsin\ ps = false\}$

33°	$isempty(tree) = false$	hyp
34°	$isvalid(norm(addpsn(addroleleaf(rl, ps), tree))) = isvalid((OR\ addsub\ norm(addroleleaf(rl, ps))\ norm(tree)))$	33°, gat:norm3
	$= topok((OR\ addsub\ norm(addroleleaf(rl, ps))\ norm(tree)))\ and$	
	$\quad evps((OR\ addsub\ norm(addroleleaf(rl, ps))\ norm(tree)))$	vt:isvalid1
	$= \{topokps(norm(addroleleaf(rl, ps)))\ and\ (topok(norm(tree))\ or\ topokps(norm(tree)))\}\ and$	
	$\quad \{isvps(norm(addroleleaf(rl, ps)))\ or\ evps(norm(tree))\}$	vt:topok4, vt:evps4
	$= \{true\ and\ (true\ or\ topokps(norm(tree)))\}\ and\ \{isvps(norm(addroleleaf(rl, ps)))\ or\ evps(norm(tree))\}$	L2, L4
	$= isvps(norm(addroleleaf(rl, ps)))\ or\ evps(norm(tree))$	
	$= allselected(addroleleaf(rl, ps))\ or\ evps(norm(tree))$	L3
35°	$isvalid(addpsn(addroleleaf(rl, ps), tree)) = isvalid(addrolleaf(rl, ps))\ or\ isvalid(tree)$	33°, gat:isvalid3
	$= isvalid(addroleleaf(rl, ps))\ or\ isvalid(norm(tree))$	4° (IH/A)
	$= isvalid(addroleleaf(rl, ps))\ or\ (topok(norm(tree))\ and\ evps(norm(tree)))$	vt:isvalid1
	$= isvalid(addroleleaf(rl, ps))\ or\ (true\ and\ evps(norm(tree)))$	L4
	$= allselected(addroleleaf(rl, ps))\ or\ evps(norm(tree))$	psn:isvalid1
36°	$isvalid(addpsn(addroleleaf(rl, ps), tree)) = isvalid(norm(addpsn(addroleleaf(rl, ps), tree)))$	34°, 35°

Table 2.16: Proof `T1`: Part `T1.1.2`

	nil	
1°	$topok(norm(emptygat(T))) = topok(emptyvt)$	gat:norm1
	$= true$	vt:topok1 — BC/A
2°	$topok(norm(tree)) = true$	IH/A
3°	$topok(norm(addpsn(emptypsn(nm), tree))) = topok(norm(tree))$	gat:addpsn1
	$= true$	2° — BC/B
4°	$topok(norm(addpsn(ps, tree))) = true$	IH/B
5°	$isempty(tree) = true \mid isempty(tree) = false$	
6°	$isempty(tree) = true$	hyp
7°	$topok(norm(addpsn(addroleleaf(rl, ps), tree))) = topok(norm(addroleleaf(rl, ps)))$	6°, gat:norm2
8°	$isempty(ps) = true \mid isempty(ps) = false$	
9°	$isempty(ps) = true$	hyp
10°	$topok(norm(addroleleaf(rl, ps))) = topok(leaf(rl))$	psn:norm2
	$= true$	vt:topok2
11°	$topok(norm(addpsn(addroleleaf(rl, ps), tree))) = true$	7°, 10°
12°	$isempty(ps) = false$	hyp
13°	$topok(norm(addroleleaf(rl, ps))) = topok((AND\ addsub\ leaf(rl)\ norm(ps)))$	psn:norm3
	$= topokps((AND\ addsub\ leaf(rl)\ norm(ps)))$	vt:topok3
	$= isleaf(leaf(rl))\ and\ topokps(norm(ps))$	vt:topokps3
	$= true$	vt:isleaf2, L2
14°	$topok(norm(addpsn(addroleleaf(rl, ps), tree))) = true$	7°, 13°
15°	$topok(norm(addpsn(addroleleaf(rl, ps), tree))) = true$	8°, 11°, 14°
16°	$isempty(tree) = false$	hyp
17°	$topok(norm(addpsn(addroleleaf(rl, ps), tree))) = topok((OR\ addsub\ norm(addrolleaf(rl, ps))\ norm(tree)))$	16°, gat:norm3
	$= topokps(norm(addroleleaf(rl, ps)))\ and\ (topok(norm(tree))\ or\ topokps(norm(tree)))$	vt:topok4
	$= true\ and\ (true\ or\ topokps(norm(tree)))$	4°, 2°
	$= true$	
18°	$topok(norm(addpsn(addroleleaf(rl, ps), tree))) = true$	5°, 15°, 17°
19°	$topok(norm(addpsn(ps, tree))) = true \Rightarrow topok(norm(addpsn(addroleleaf(rl, ps), tree))) = true$	4°, 18°
20°	$\forall(ps \in psn) \bullet topok(norm(addpsn(ps, tree)))$	4°, 19°, ind
21°	$topok(norm(tree)) = true \Rightarrow \forall(ps \in psn) \bullet topok(norm(addpsn(ps, tree)))$	2°, 20°
22°	$\forall(tree \in gat) \bullet topok(norm(tree)) = true$	1°, 21°, ind.

Table 2.17: Proof T1: Lemma L4

Theorem $T2$

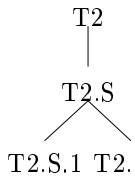

Figure 2.3: T2: Dependency Tree

The deductions corresponding to the lines 40° respectively 50° of the proof shown in Table 2.21 need an explanation. Let us consider the hypothesis in line 37° in Table 2.19. The hypothesis actually constrains the variables v' and v *implicitly*. However, in order to be able to conclude $isvalid(l\ addsub\ v'\ v) = false$, we need to know the values of v' and v, which means that we have to make them explicit. For this, we consider all possible terms $norm(G)\ where\ G = addpsn(addroleleaf(rl, ps), g)$ can evaluate to. These are:

1. $leaf(r)$
2. $(AND\ addsub\ leaf(rl)\ norm(ps))$ and $isempty(g) = true$
3. $(OR\ addsub\ norm(addroleleaf(rl, ps)), norm(g))$

The term $leaf(rl)$ is not of interest because the hypothesis in line 37° syntactically forbids it. We have a similar situation for the case where $l = AND$ and for the case where $l = OR$:

For the case where $l = AND$, the hypothesis forbids term 3. So in this case we have to consider term 2. It tells us that $(v' = leaf(rl)) = false$ or $(v = norm(ps)) = false$, if the hypothesis is not to be invalidated.

For the case where $l = OR$, the hypothesis forbids term 2, which means that we have to consider term 3. It tells us that $(v' = norm(addroleleaf(rl, ps))) = false$ or $(v = norm(g)) = false$, if the hypothesis is not to be invalidated.

	nil	
1°	$\neg\forall(v \in vt) \bullet (emptyvt = v) = false \Rightarrow isvalid(v) = false$	hyp
2°	$\exists(v \in vt) \bullet (emptyvt = v) = false \ \& \ isvalid(v) = true$	1°
3°	$(emptyvt = emptyvt) = false \ \& \ isvalid(emptyvt) = true$	hyp
4°	$(emptyvt = emptyvt) = true \ \& \ isvalid(emptyvt) = false$	vt:isvalid1, vt:evps1
5°	$\bot$	3°, 4°
6°	$\bot$	2°, 5°
7°	$\forall(v \in vt) \bullet (emptyvt = v) = false \Rightarrow isvalid(v) = false$	1°, 6°
8°	$\forall(v \in vt) \bullet (norm(emptygat(T)) = v) = false \Rightarrow isvalid(v) = false$	7°, gat:norm1 — BC/A
9°	$\forall(v \in vt) \bullet (norm(g) = v) = false \Rightarrow isvalid(v) = false$	IH/A
10°	$\forall(v \in vt) \bullet (norm(addpsn(emptypsn(nm), g)) = v) = false \Rightarrow isvalid(v) = false$	gat:addpsn1 — BC/B
11°	$\forall(v \in vt) \bullet (norm(addpsn(ps, g)) = v) = false \Rightarrow isvalid(v) = false$	IH/B
	$\vdots$	Table 2.19
76°	$\forall(v \in vt) \bullet (norm(addpsn(addroleleaf(rl, ps), g)) = v) = false \Rightarrow isvalid(v) = false$	
77°	$\forall(v \in vt) \bullet (norm(addpsn(ps, g)) = v) = false \Rightarrow isvalid(v) = false$	
	$\Rightarrow \forall(v \in vt) \bullet (norm(addpsn(addroleleaf(rl, ps), g)) = v) = false \Rightarrow isvalid(v) = false$	11°, 76°
78°	$\forall(ps \in psn)\forall(v \in vt) \bullet (norm(addpsn(ps, g)) = v) = false \Rightarrow isvalid(v) = false$	10°, 77° ind
79°	$\forall(v \in vt) \bullet (norm(g) = v) = false \Rightarrow isvalid(v) = false$	
	$\Rightarrow \forall(ps \in psn)\forall(v \in vt) \bullet (norm(addpsn(ps, g)) = v) = false \Rightarrow isvalid(v) = false$	9°, 78°
80°	$\forall(g \in gat)\forall(v \in vt) \bullet (norm(g) = v) = false \Rightarrow isvalid(v) = false$	8°, 79°, ind.

Table 2.18: Proof T2: outer

	$\{IH/A, IH/B\}$	
12°	$(norm(G) = emptyvt) = false$	hyp
13°	$isvalid(emptyvt) = false$	vt:isvalid1, vt:evps1
14°	$(norm(G) = emptyvt) = false \Rightarrow isvalid(emptyvt) = false$	1°, 2° — BC/1
15°	$(norm(G) = leaf(r)) = false$	hyp
	⋮	Table 2.20
32°	$isvalid(leaf(r)) = false$	
33°	$(norm(G) = leaf(r)) = false \Rightarrow isvalid(leaf(r)) = false$	4°, 32° — BC/2
34°	$(norm(G) = v) = false \Rightarrow isvalid(v) = false$	IH
35°	$v' \in vt$	hyp
36°	$l \in label$	hyp
37°	$(norm(G) = (l\ addsub\ v'\ v)) = false$	hyp
38°	$l = AND \mid l = OR$	
39°	$l = AND$	hyp
	⋮	Table 2.21
48°	$isvalid((l\ addsub\ v'\ v)) = false$	
49°	$l = OR$	hyp
	⋮	Table 2.21
68°	$isvalid((l\ addsub\ v'\ v)) = false$	
69°	$isvalid((l\ addsub\ v'\ v)) = false$	38°, 48°, 68°
70°	$(norm(G) = (l\ addsub\ v'\ v)) = false \Rightarrow isvalid((l\ addsub\ v'\ v)) = false$	37°, 69°
71°	$l \in label \Rightarrow (norm(G) = (l\ addsub\ v'\ v)) = false \Rightarrow isvalid((l\ addsub\ v'\ v)) = false$	36°, 70°
72°	$\forall(l \in label) \bullet (norm(G) = (l\ addsub\ v'\ v)) = false \Rightarrow isvalid((l\ addsub\ v'\ v)) = false$	71°
73°	$v' \in vt \Rightarrow \forall(l \in label) \bullet (norm(G) = (l\ addsub\ v'\ v)) = false \Rightarrow isvalid((l\ addsub\ v'\ v)) = false$	35°, 72°
74°	$\forall(v' \in vt)\forall(l \in label) \bullet (norm(G) = (l\ addsub\ v'\ v)) = false \Rightarrow isvalid((l\ addsub\ v'\ v)) = false$	73°
75°	$(norm(G) = v) = false \Rightarrow isvalid(v) = false$	
	$\Rightarrow \forall(v' \in vt)\forall(l \in label) \bullet (norm(G) = (l\ addsub\ v'\ v)) = false \Rightarrow isvalid((l\ addsub\ v'\ v)) = false$	34°, 74°
76°	$\forall(v \in vt) \bullet (norm(G) = v) = false \Rightarrow isvalid(v) = false$	14°, 33°, 75°, ind

Table 2.19: Proof T2: Part T2.S with $G = addpsn(addroleleaf(rl, ps), g)$

$\{IH/A, IH/B\}$

15°	$(norm(addpsn(addroleleaf(r, ps), g)) = leaf(r)) = false$	hyp
16°	$isempty(g) = false \mid isempty(ps) = false$	15°
17°	$isempty(g) = false$	hyp
18°	$norm(addpsn(ps, g)) = (OR\ addsub\ norm(ps)\ norm(g))$	17°, gat:norm3
19°	$(norm(addpsn(ps, g)) = leaf(r)) = false$	18°
20°	$isvalid(leaf(r)) = false$	11°, 19°
21°	$isempty(ps) = false$	hyp
22°	$isempty(g) = true \mid isempty(g) = false$	
23°	$isempty(g) = true$	hyp
24°	$norm(addpsn(ps, g)) = norm(ps)$	23°, gat:norm2
25°	$(norm(addpsn(ps, g)) = leaf(r)) = false$	24°
26°	$isvalid(leaf(r)) = false$	11°, 25°
27°	$isempty(g) = false$	hyp
28°	$norm(addpsn(ps, g)) = (OR\ addsub\ norm(ps)\ norm(g))$	27°, gat:norm3
29°	$(norm(addpsn(ps, g)) = leaf(r)) = false$	28°
30°	$isvalid(leaf(r)) = false$	11°, 29°
31°	$isvalid(leaf(r)) = false$	22°, 26°, 30°
32°	$isvalid(leaf(r)) = false$	16°, 20°, 31°

Table 2.20: Proof `T2`: Part `T2.S.1`

$\{IH/A, IH/B, IH, v' \in vt, l \in label, (norm(addpsn(addroleleaf(rl, ps), g)) = (l\ addsub\ v'\ v)) = false\}$

39°	$l = AND$	hyp
40°	$(v' = leaf(rl)) = false \mid ((v = norm(ps)) = false\ \&\ isempty(g) = true)$	see comments
41°	$(v' = leaf(rl)) = false$	hyp
42°	$isleaf(v') = false$	41°
43°	$isvalid((l\ addsub\ v'\ v)) = false$	42°
44°	$(v = norm(ps)) = false\ \&\ isempty(g) = true$	hyp
45°	$\forall(v \in vt) \bullet (norm(ps) = v) = false \Rightarrow isvalid(v) = false$	11°, 44°, gat:norm2
46°	$isvalid(v) = false$	44°, 45°
47°	$isvalid((l\ addsub\ v'\ v)) = false$	46°
48°	$isvalid((l\ addsub\ v'\ v)) = false$	40°, 43°, 47°
49°	$l = OR$	hyp
50°	$(v' = norm(addroleleaf(rl, ps))) = false \mid (v = norm(g)) = false$	see comments
51°	$(v' = norm(addroleleaf(rl, ps))) = false$	hyp
52°	$isempty(ps) = true \mid isempty(ps) = false$	
53°	$isempty(ps) = true$	hyp
54°	$norm(addroleleaf(rl, ps)) = leaf(rl)$	53°, psn:norm2
55°	$(v' = leaf(rl)) = false$	51°, 54°
56°	$isleaf(v') = false$	55°
57°	$isvalid((l\ addsub\ v'\ v)) = false$	56°
58°	$isempty(ps) = false$	hyp
59°	$norm(addroleleaf(rl, ps)) = (AND\ addsub\ leaf(rl)\ norm(ps))$	58°, psn:norm3
60°	$(v' = (AND\ addsub\ leaf(rl)\ norm(ps))) = false$	51°, 59°
61°	$\exists(x \in vt)\exists(y \in vt) \bullet (v' = (AND\ addsub\ x\ y)) = true\ \&\ ((x = leaf(rl)) = false \mid (y = norm(ps)) = false)$	60°
62°	$isleaf(v') = false$	61°
63°	$isvalid((l\ addsub\ v'\ v)) = false$	62°
64°	$isvalid((l\ addsub\ v'\ v)) = false$	52°, 57°, 63°
65°	$(v = norm(g)) = false$	hyp
66°	$isvalid(v) = false$	65°, 9°
67°	$isvalid((l\ addsub\ v'\ v)) = false$	66°
68°	$isvalid((l\ addsub\ v'\ v)) = false$	50°, 64°, 67°

Table 2.21: Proof T2: Part T2.S.2

Chapter 3

Implementation

MIERA has been implemented in the Java programming language using JDK 1.2.3. In particular, the packages `java.security` and `java.security.cert` have been used to implement the security related aspects of MIERA. Hashes are created using the SDA-1 function. The implementation does not offer any certificate management.

3.1 Architectural Overview

The package `core` gives a possible implementation of the essential features of MIERA. The package `demo` then, provides an interactive demonstrator for these features. As Table 3.1 shows, `core` consists of the two main parts `authorisation tree` and `transaction`. The package `authorisationtree` groups together all the classes needed to model the different kinds of authorisation trees that intervene in MIERA+. The package `transaction` groups together classes that implement the concept of a transaction as it is understood in the MIERA scheme. The package `demo` consists of the two main parts `model` and `view`.

3.2 Core

3.2.1 Authorisation Tree Classes

3.2.1.1 `GeneralAuthorizationTree`

The `GeneralAuthorizationTree`, *GAT*, class implements an authorisation tree as it is described in [1]. It is a node, i.e. it can essentially be represented by an ordered pair, where the first member, called *label field*, denotes the transaction type, let's say T, the authorisation tree corresponds to, and where the second member, called *descendants field*, references the set of permission set nodes `permissionSetNodes` that are required to authorise a transaction of type T. The transaction type field corresponds to the root level and the permission set field corresponds to the intermediate level in the authorisation tree of [1].

`permissionSetNodes` is an instance of the class `Authorization`, which basically is a set of instances of the class `PermissionSetNode`. `PermissionSetNode`

miera	core	authorizationtree	*Authorization.java* *BinaryAuthorizationTree.java* *BinaryTreeNode.java* *GeneralAuthorizationTree.java* *MultiTreeNode.java* *ObscureAuthorizationTree.java* *PermissionSet.java* *PermissionSetNode.java* *PublishableAuthorizationTree.java* *Role.java* *ScannableStack.java* *TransmittableAuthorizationTree.java* *TreeNode.java* *VerificationTree.java*
		doc	
		error.log	
		transaction	*Container.java* *FormContainer.java* *FormLog.java* *Transaction.java* *TransactionLog.java* *TreeContainer.java* *TreeLog.java* *VerificationLog.java*
	demo	*Demo.java*	
		doc	
		error.log	
		model	see Table 3.2
		view	see Table 3.3
	error.log		

Table 3.1: Components: Global View

has a structure analogue to *GAT* but with the label denoting the name of the permission set its descendants field references, namely `roles`.

Thus `roles` is an instance of the class `PermissionSet`, which basically is a set of instances of the class `Role`, which contains as the essential information the role name. (In fact, the demonstrator described in Section 3.3 exclusively handles role names, i.e. strings, and not instances of `Role`. In a real application however, there might be a need of having a special class to model roles, which is why `core` provides this special class.) Roles correspond to leaves in the authorisation tree of [1].

The class *GAT* offers various methods for construction and manipulation of its instances. Tasks that would (and in fact are in the demonstrator) typically carried out by the transaction administrator of the company that uses the MIERA scheme. The transaction administrator serialises instances of *GAT* into a data base, where they can be retrieved by requestors wishing to initiate a transaction. Requestors then call the method `getNormalForm`, which transforms an instance of *GAT* into an instance of `PublishableAuthorizationTree` as it is illustrated by Figure 1.2.

3.2.1.2 `PublishableAuthorizationTree`

A `PublishableAuthorizationTree`, *PAT*, is a binary node, i.e. a node where the descendants field references exactly two other nodes, namely `left` and `right`, which are also instances of the class *PAT* (see Figure 3.1).

Instances of *PAT* are created by calling the method `getNormalForm` on an instance of the class *GAT*. The resulting tree can take two kinds of labels, namely `AND` and `OR`. A tree node takes the label `AND` iff what derives from the node corresponds to a subset of a permission set, otherwise it takes the label `OR` (see Figure 1.2).

Instances of *PAT* are used by the transaction administrator and by requestors. The transaction administrator calculates the *reference hash* from each instance and makes it available in a public data base for later authorisation verification. Requestors call the method `getObscureForm` on instances of *PAT* indicating the set of roles they have chosen to sign the transaction form. `getObscureForm` hashes the selected role names and the permission sets not involved in the current transaction request. The resulting object is an instance of `TransmittableAuthorizationTree` (see Figure 1.2).

3.2.1.3 `TransmittableAuthorizationTree`

A `TransmittableAuthorizationTree`, *TAT*, is, just like a *PAT*, a binary node (they have a common super class, see Figure 3.1) but with the difference that the label of a certain node n can also contain a value that results from hashing the nodes that derive from n rather than just the values `AND` and `OR`.

The *TAT* offers the method `getReconstructableForm`, which is typically called by the requestor, and creates an intermediate representation, *IR*, of an instance of the class (see Figure 1.2). Using EBNF-notation, *IR* can be described as follows (blanks and line breaks do not have any significance):

```
<ReconstructableForm>::= <MetaInformation>
            <MetaSeparatorCharacter>
            <PostFixNotationOfTransmittableAuthorizationTree>
```

```
<MetaInformation>::= <MetaSeparatorCharacter>
             <EscapeCharacter>
             <ObjectSeparatorCharacter>
                 <ConcHashCharacter>
                 <HashFunctionName>
             <MetaSeparatorCharacter>
                 <RoleNameHashes>

<RoleNameHashes>::= <EscapedRoleNameHash>
            { <ObjectSeparatorCharacter> <EscapedRoleNameHash> }

<PostFixNotationOfTransmittableAuthorizationTree>::= <PostFixNotationOfLeaf>
                             | <PostFixNotationOfNode>

<PostFixNotationOfLeaf>::= <EscapedHash>

<EscapedHash>::= <EscapedRoleNameHash> | <EscapedSubTreeHash>

<PostFixNotationOfNode>::= (<PostFixNotationOfLeaf> | <PostFixNotationOfNode>)
               <ObjectSeparatorCharacter>
               (<PostFixNotationOfLeaf> | <PostFixNotationOfNode>)
               <ConcHashCharacter>
```

The encoding naturally contains meta symbols, namely

- $<MetaSeparatorCharacter>$
- $<EscapeCharacter>$
- $<ObjectSeparatorCharacter>$
- $<ConcHashCharacter>$

which have to be escaped in the actual hash values because the hash function might produce such symbols. In the current implementation the hash function is made to return a string of 20 small hexadecimal digits and the meta symbols have (for illustration purposes) been chosen to be ';', 'e', ',' and '#' respectively.

$<MetaSeparatorCharacter>$, $<ObjectSeparatorCharacter>$ and $<ConcHashCharacter>$ are *punctuation symbols*, which, according to the definition of the term, indicate positions of synchronisation in the IR string. An analyser-interpreter would typically interpret the meaning of a parsed substring at such positions (and only at such). $<ConcHashCharacter>$ tells the analyser-interpreter when to synthesise a node from two tokens interpreted as subtrees (leaves or proper subtrees).

The meta information contains the meta symbols, the name of the used hash function ('SHA' in our case) and the hashes of the selected role names.

3.2.1.4 `VerificationTree`

The `VerificationTree`, VT, has the same structure as TAT, however offers other methods. There is a constructor method and a method that calculates its hash value. Both methods are called indirectly (through a call of `verify`, see Section 3.2.2.1) by the verifier.

The constructor takes as input parameter a so called *scannable stack* that must hold IR. A scannable stack is a stack where 'pop' pops symbols off the stack until a so called *end marker* lies at the top of the stack, or yet the stack

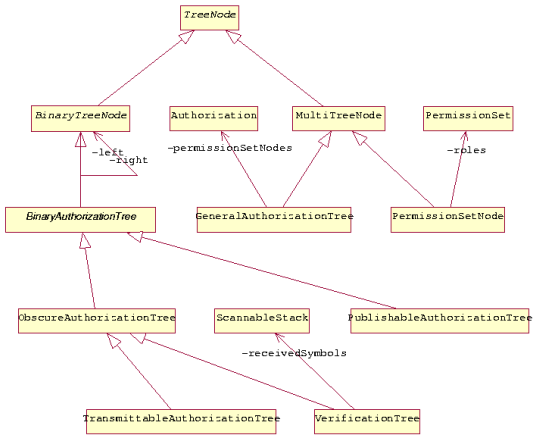

Figure 3.1: Authorisation Tree Classes

is empty. The end markers then, are in fact identical to the in Section 3.2.1.3 mentioned meta symbols.

In the construction of VT based on IR, two scannable stacks are used as lexical analysers, each one at a different level of abstraction. The first is passed to the constructor and is used with <*MetaSeparatorCharacter*> as an end marker in order to extract the different pieces of meta information and the post-fix encoding of TAT based on IR. The second is used with <*EscapeCharacter*>, <*ObjectSeparatorCharacter*> and <*ConcHashCharacter*> as end markers to extract the role name hashes from IR. A syntactic analyser and interpreter then uses the role name hash tokens to actually construct VT.

3.2.2 Transaction Classes

3.2.2.1 `Transaction`

The class `Transaction` implements the concept of a transaction as it is defined in Table 1.2. It has a sender name attribute and an instance of the class references three instances corresponding to the classes `FormContainer`, `TreeContainer` and `TransactionLog` (see Figure 3.2).

The form container holds all signature related authorisation information and the tree container holds all the authorisation information that is related to the authorisation tree.

The class offers the single method `verify`, which is called directly by the verifier and checks if a transaction — an instance of `Transaction` — has been correctly authorised. `verify` calls a method of the same name on the form container and on the tree container. Also, it checks if the role name hashes extracted from the form container and the ones extracted from the tree container match. (The role name hashes are extractable only after both containers have gone through the verification procedure.) The check ensures that the identities of the effective signers and the identities of the signers that have been declared

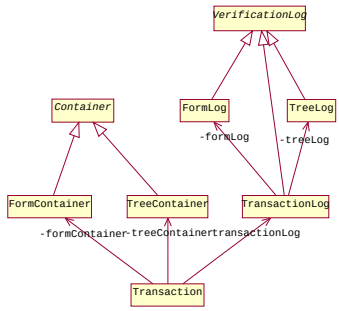

Figure 3.2: Transaction Classes

by the requestor as such are in fact the same.

The transaction log references an instance of the class `FormLog` and an instance of the class `TreeLog`. It keeps book of the results of the authorisation verification operations that are performed at the level of the form and tree container and of the ones that are performed at the level of the transaction itself.

3.2.2.2 `FormContainer`

The class `FormContainer` references four instances corresponding to the classes `FormLog`, `SignedObject`, `Form` and `LinkedList`. The form log keeps book of the results of the authorisation verification operations that are performed on the form. The signed object contains the signed transaction form, and the field referencing the form object will contain the unsigned transaction form once the signatures on it have been verified. The instance of `LinkedList` contains the anonymous role certificates of the signers in the inverse order they have signed the transaction form. The signatures on the transaction form are checked against the public keys contained in these certificates.

In the (common) case where the transaction form bears more than one signature, the signed object encapsulates another signed object and so forth until the unsigned transaction form is reached. This packing mechanism implements the signing policy adopted in the MIERA scheme, which says that signatures are placed one on top of the other. As a consequence, the signature verification process has to unpack the unsigned transaction form in order to be able to reference it.

The verification is performed by the method `verify`, which calls the methods `verifyAnonymousRoleCertificates` and `verifyFormSignatures`. The first takes the public key of the certificate authority of the sender of the transaction as input parameter and checks the anonymous role certificates for authenticity and integrity. The second performs the signature verification as described in the previous paragraph.

3.2.2.3 `TreeContainer`

The class `TreeContainer` references four instances corresponding to the classes `TreeLog`, `String`, `VerificationTree` and `SignedObject`. The tree log keeps book of the results of the authorisation verification operations that are per-

formed on the authorisation tree. The string field holds the intermediate representation IR of the authorisation tree as it is described in Section 3.2.1.3. From IR the verifier constructs the verification tree vt, which is stored in the verification tree field. The signed object contains the signed reference hash rh, which, for simplicity, is sent with the transaction rather than letting the verifier fetch it in a public data base at the sender company.

The verification is performed by the method `verify`, which calls the methods `verifyReferenceHash` and `rootHashesAreEqual`. The first takes the public key of the certificate authority of the sender of the transaction as input parameter and checks the reference hash for authenticity and integrity. The second performs the equality check between the hash calculated from rh and the hash calculated from vt.

3.2.2.4 `TransactionLog`

An instance of the class `TransactionLog` groups together an instance of `FormLog` and an instance of `TreeLog`, which are instantiated and updated with the verification results in the respective containers. The fields of the form log are: `anonymousRoleCertificatesValidity` and `formSignaturesValidity`. The ones of the tree log are: `referenceHashValidity` and `rootHashesEquality`. All fields are of type `boolean`. The transaction log is used as a back-up of the verification results.

3.3 Demonstrator

The demonstrator is an application[1], which consists of two packages `model` and `view` (see Table 3.1) and runs on a single machine. As the name indicates, `model` defines the data model for the demonstrator. `view` contains an implementation of a graphical user interface (*GUI.java*), which illustrates the features built into the model and allows interactive access by the user. In fact, `view` references those objects that have previously been instantiated by `model` from classes defined in `core`. Figure 3.3 illustrates the architecture of the demonstrator.

3.3.1 Model Classes

The class `DemoModel` constitutes the model for the demonstrator. An instance of the class essentially references an instance of the class `Client` and an instance of the class `Provider` (see Figure 3.4). `DemoModel` is a subclass of the abstract class `Model`. The model has been tested independently from the view, which made necessary the simulation of the operations the view would normally perform on the model. For this purpose `DemoModel` has been subclassed by `TestModel`, which provides these operations. Analogous subclasses have been created for the classes `Client`, `Requestor` and `Provider` for the same purpose (see Figures 3.5 respectively 3.6). The class `ModelTester` then performs the simulation and by doing so tests the model for correct functioning.

[1] run by executing the code `java com.ibm.miera.demo.Demo` in the directory `miera`.

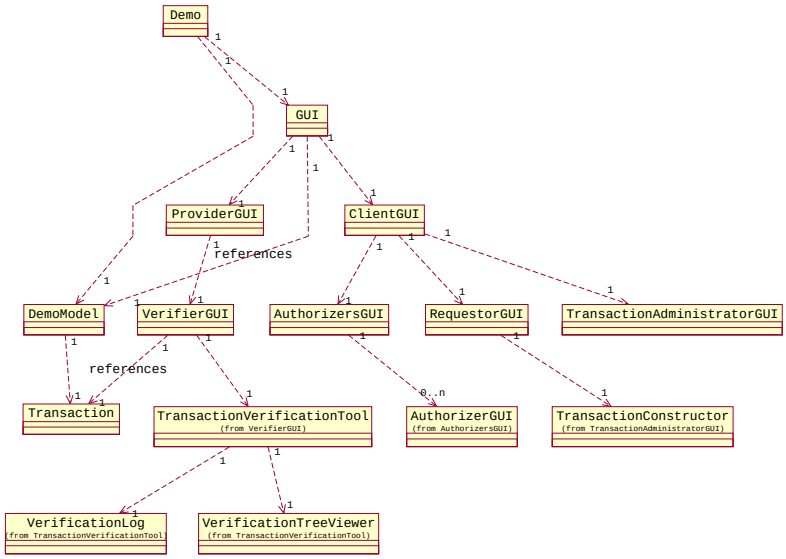

Figure 3.3: The `Demo` Class

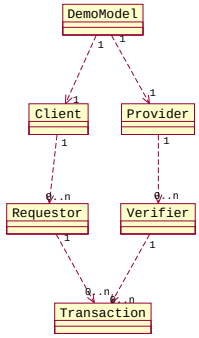

Figure 3.4: The `DemoModel` Class

model	*DemoModel.java* *Model.java* *ModelTester.java* *TestModel.java*		
	communication	*channel.txt*	
	db	some classes abstracting the authorisation tree and the anonymous role certificate data base.	
	forms	various empty transaction forms (HTML files)	
	transactors	*Transctor.java*	
		client	*Authorizer.java* *CertificateAuthority.java* *Client.java* *ClientEntity.java* *ModelTestClient.java* *ModelTestRequestor.java* *Requestor.java* *RoleAdministrator.java* *SignatureCollector.java* *Signer.java* *TransactionAdministrator.java*
		provider	*ModelTestProvider.java* *Provider.java* *Verifier.java*

Table 3.2: Model Components

3.3.1.1 Client Classes

The class `Client` models CLIENT and groups together a certificate authority, a role administrator, a transaction administrator, a set of employees with authorisation competence and a set of employees that may act as transaction requestors. The class offers the single method `send`, which takes as parameter the transaction to be send to PROVIDER. The transaction is serialised into the file `channel.txt` (see Table 3.2), which models the communication channel between CLIENT and PROVIDER.

The class `CertificateAuthority` offers methods to sign the reference hash and the anonymous role certificates. A sample of anonymous role certificates has been created for demonstration purposes using the *keytool* facility under UNIX. *keytool* creates X509v1 certificates [2]. For example, the anonymous role certificate for the possible authoriser *Luke O'Connor* with the unique relative role name *AppliedComputerScience:RSM:NetworkSecurity&Cryptography Luke J. O'Connor*, let's call it *urrn*, has been created executing the following command:

```
keytool -genkey
    -alias 6012a792c0c128700ed3eb7df86c176e718fedb6
    -keyalg DSA
    -keysize 1024
    -sigalg SHA1withDSA
    -dname "CN=6012a792c0c128700ed3eb7df86c176e718fedb6,
        OU=Research Division,
        O=IBM,
        L=Zurich,
        S=Zurich,
        C=CH"
    -keypass mykeypass
    -validity 365
    -keystore D:\com\ibm\miera\demo\model\db\.keystore
    -storepass mystorepass
```

where

- `.keystore` denotes the data base where this anonymous role certificate is going to be stored
- `mystorepass` is the pass word used to protect the integrity of `.keystore`
- `mykeypass` is the pass word which the authoriser Luke O'Connor would use to retrieve his private key [3] from `.keystore`
- `6012a792c0c128700ed3eb7df86c176e718fedb6` is the hash value of *urrn*
- `SHA1withDSA` denotes the algorithm for signature creation
- `DSA` denotes the algorithm for key pair generation

[2] This is a difference to [1], where X509v3 certificates have been suggested to use.
[3] His signing key.

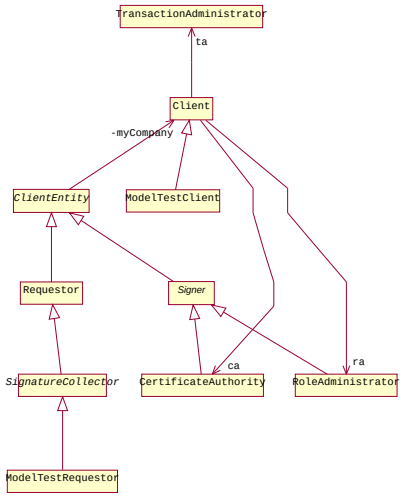

Figure 3.5: Client Classes

The class `Role Administrator` offers a method that takes as parameter a role name and returns the corresponding identity of the owner of that role [4]. In the current implementation, a role always has only one owner because role names are modelled by *unique relative role names* as described in Table 1.2. The method is called by the requestor, who has to find out whom to submit the transaction form for signing to.

The class `Transaction Administrator` offers methods to load from, store into and delete in the corresponding data base authorisation trees.

The class `Requestor` offers methods to retrieve, first, the identities of role owners and, second, the corresponding anonymous role certificates.

The class `Authorizer` offers the single method `sign`, which takes as parameter an instance of `Transaction`.

3.3.1.2 Provider Classes

The class `Provider` has two attributes: one that references the list of public keys of certificate authorities that PROVIDER recognises as such, and another one that references the set of employees that are allowed to verify an incoming transaction. The transaction is read from the communication channel by the method `receiveTransaction`.

The class `Verifier` offers the method `verifyTransaction`, which takes as parameter the transaction which the authorisation is to be verified of.

3.3.2 View Classes

The view package consists of the two main parts `client` and `provider`, which implement a graphical user interface (abbreviated *GUI*) for the corresponding parts in the model, and last but not least (at all) the package contains the main

[4] This method implements a *backward reference* mechanism, although a very simplistic compared to the one suggested in [1].

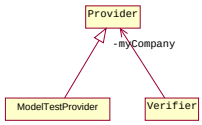

Figure 3.6: Provider Classes

GUI for the demonstrator (file *GUI.java*). Figures 3.3 and 3.7 give an overview of the logical structure of `view` and Table 3.3 lists its components. The graphical user interface has been implemented using the `javax.swing` package.

The client package contains classes that provide a GUI, first, for CLIENT itself, and , second, for each one of the users that are involved in the authorisation process taking place at CLIENT. These users are: the transaction administrator (edits the authorisation tree for each transaction type), the requestor(s) and the authoriser(s).

The provider package contains classes implementing a GUI for some verifier at PROVIDER and a GUI for PROVIDER itself.

The various classes grouped together in `client` and `provider`, let's call them primary view components, have not been defined within the scope of the main GUI in order to allow their arrangement in an environment different to the one that is created by the present main GUI. For example, it might be desirable to demonstrate MIERA on different machines, i.e. one machine simulating CLIENT and another simulating PROVIDER. The view components that *constitute* the primary view components however, have been implemented as inner classes of the latter because it would not make sens to use them in an other context.

The secondary view classes do not represent any greater interest for themselves, so we will not address them further but rather show what they produce (Figures 3.8 to 3.16). The figures reflect the order in which the different MIERA tasks are supposed to occur, that is, starting at CLIENT:

1. creation of authorisation trees by the transaction administrator (Figure 3.8)
2. retrieval of some authorisation tree and subsequent role selection by some requestor (Figure 3.9)
3. signature stamping by the selected transaction authorisers (Figure 3.10)
4. completion of the corresponding transaction form by the requestor (Figures 3.11 and 3.12) [5]
5. dispatch of the transaction to PROVIDER (Figure 3.13)
6. verification of the transaction authorisation by some verifier at PROVIDER (Figures 3.14 and 3.15)

[5]The signature collecting mechanism is being simulated in the demonstrator, i.e. `demo`. (It would have been unacceptable, for example, to simulate it in `core` since signature collecting does clearly not make part of the MIERA core functionalities.) The transaction form is passed to one signer after the other, which would make easy the implementation of *conditional signatures*. (A conditional signature on a document is a signature that is only placed on the document if a certain set of other signatures have already been placed on it.)

view	*GUI.java*	
	client	*AuthorizersGUI.java* *ClientGUI.java* *RequestorGUI.java* *TransactionAdministratorGUI.java*
	icons	icons used in the graphical user interface
	provider	*ProviderGUI.java* *VerifierGUI.java*
	utilities	various utility classes for the construction of the graphical user interface

Table 3.3: View Components

The graphical user interface of the demonstrator has been designed to illustrate the logical structure of and the flow of tasks in the MIERA scheme. The demonstrator is not a real world application and therefore not much attention has been paid to ergonomics of use.

Window nesting has been used to illustrate hierarchical relations between the GUI entities. The flow of tasks is oriented from the top to the bottom of the GUI with juxtaposition of GUI entities expressing the possibility of parallel execution of tasks.

The example transaction form is the one of a travel request, which IBM employees have to fill out before they go on business trips. This form is internal to IBM and the example provider KUONI would not be interested in this form when selling a business trip arrangement to IBM. In this sense the example is not very realistic but has been chosen to provide a familiar case for demonstration purposes at IBM.

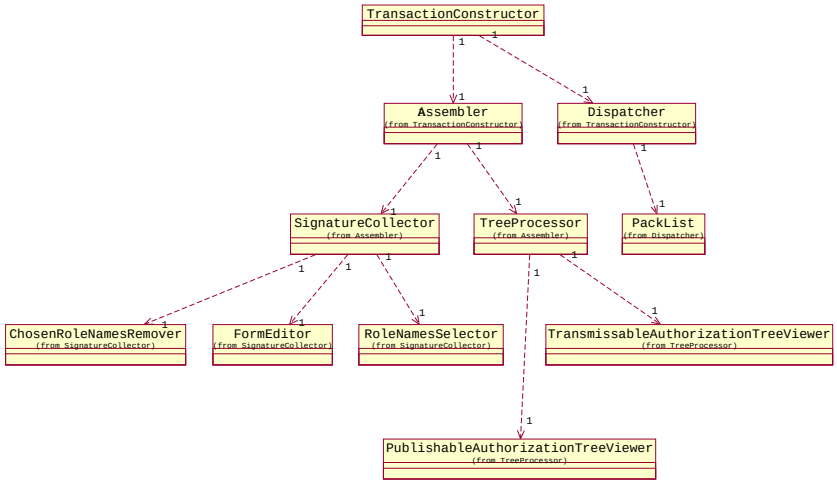

Figure 3.7: The `TransactionConstructor` Class

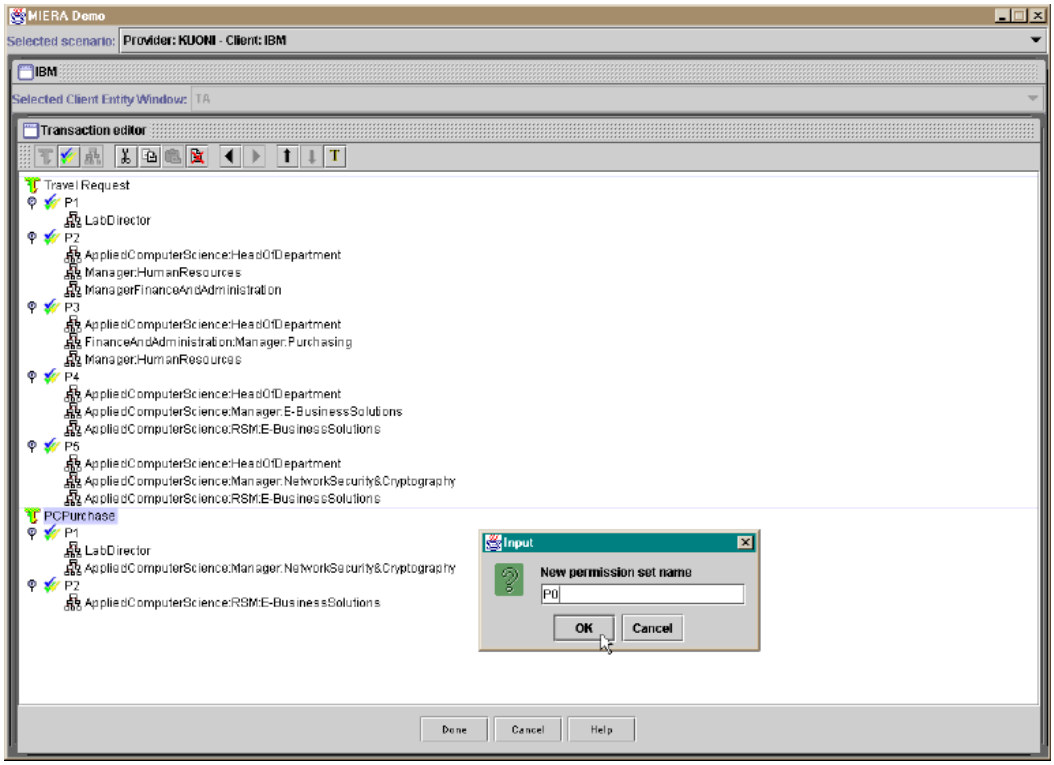

Figure 3.8: Transaction Authority GUI: Editing Authorisation Trees

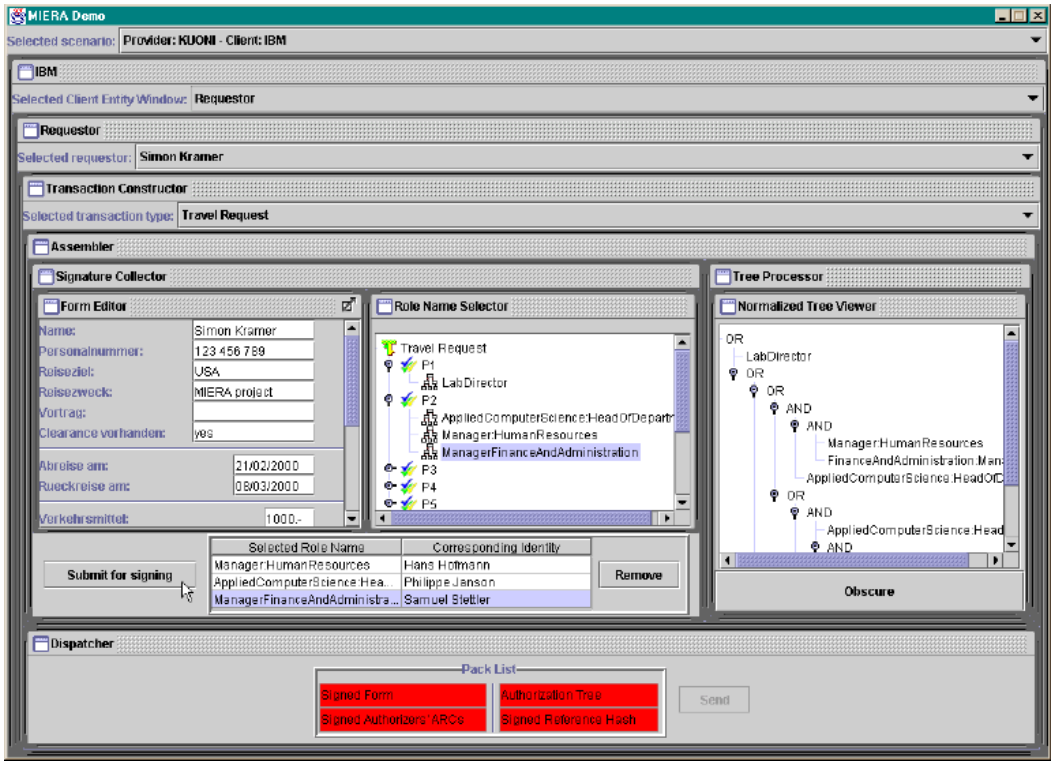

Figure 3.9: Requestor GUI: Filling out the Transaction Form and Selecting the Roles

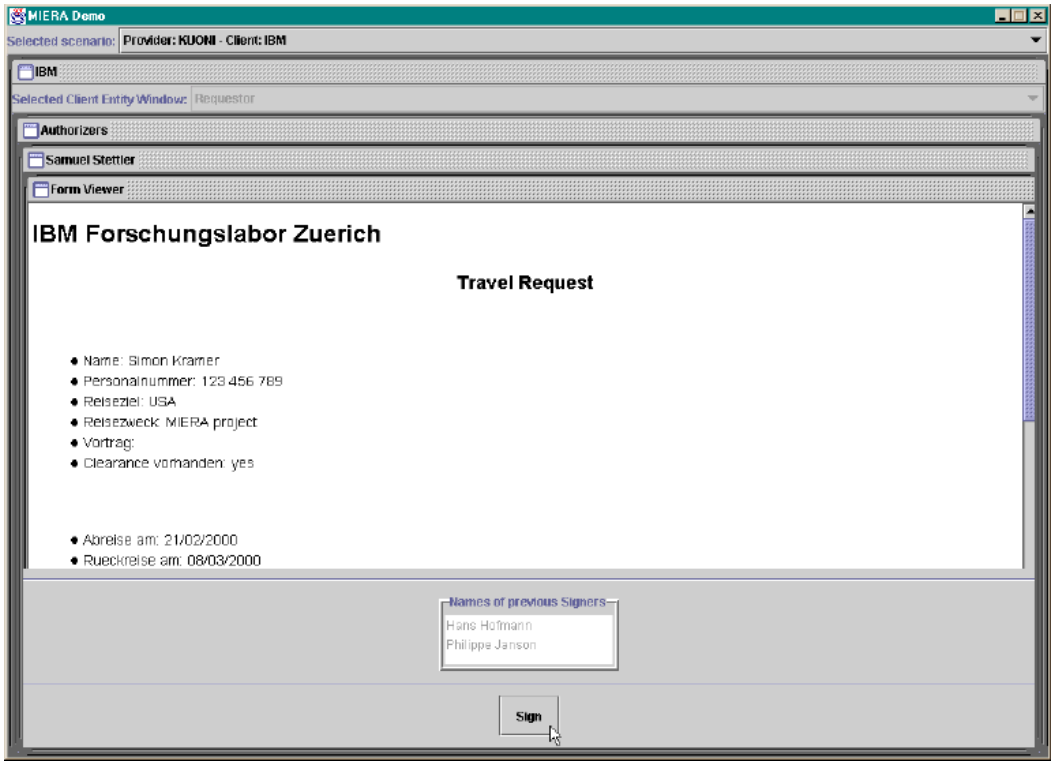

Figure 3.10: Authorisers GUI: Signing the Transaction Form

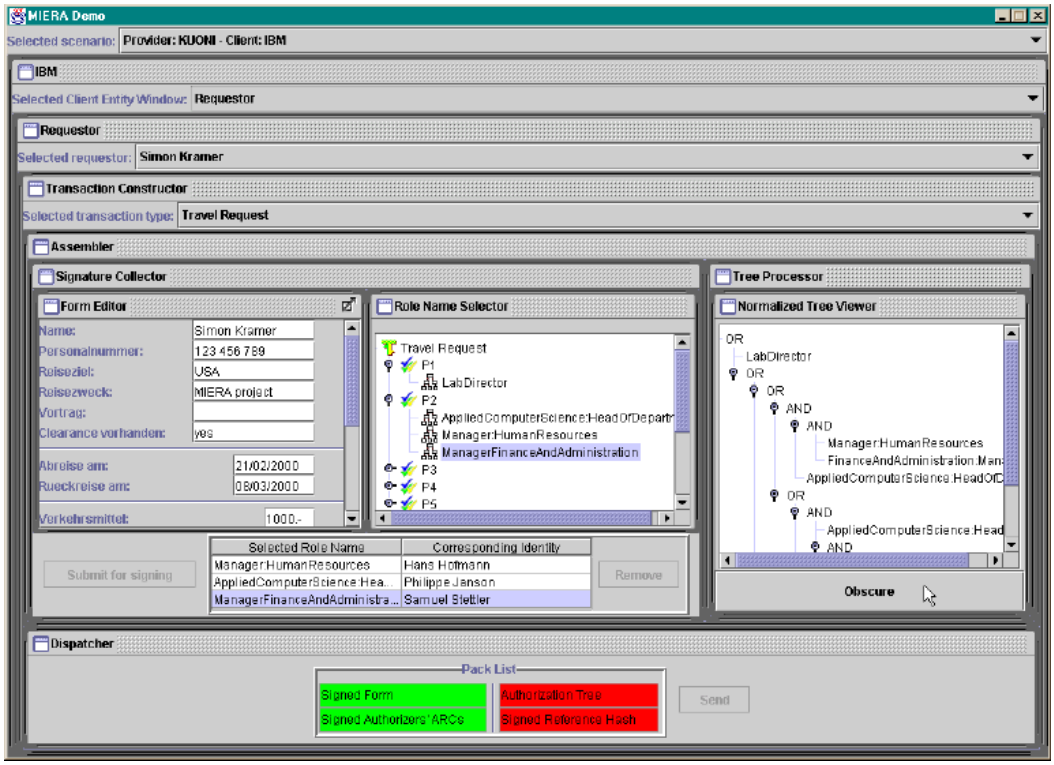

Figure 3.11: Requestor GUI: Obscuring the Authorisation Tree

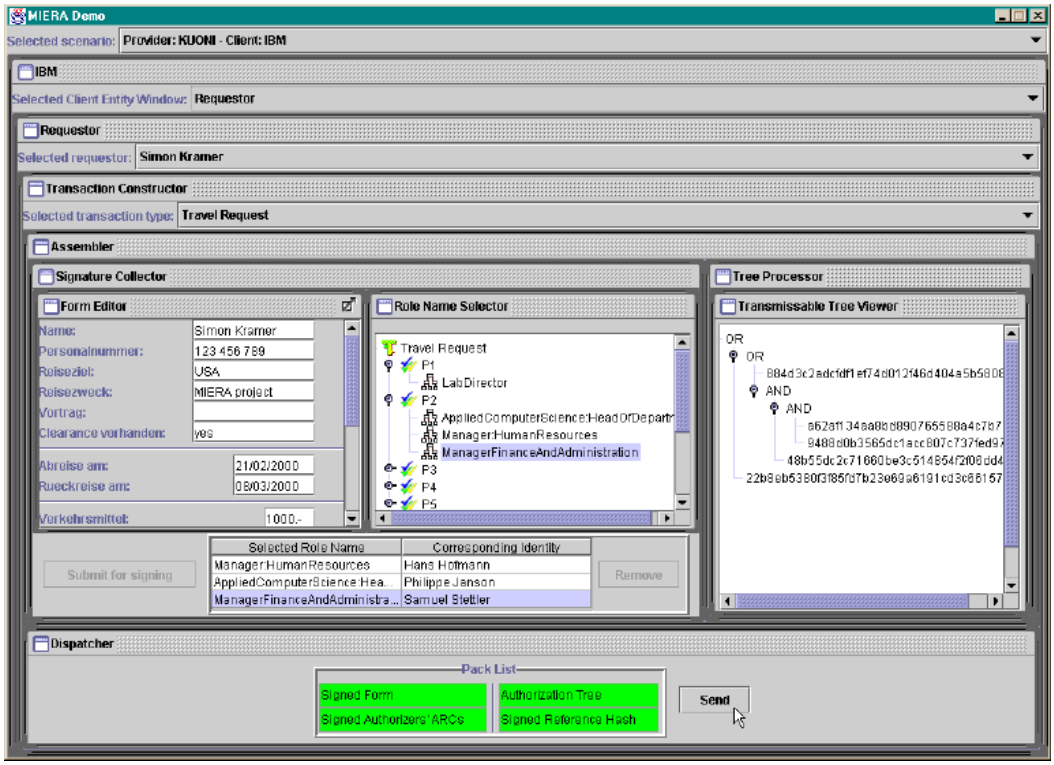

Figure 3.12: Requestor GUI: Dispatching the Transaction

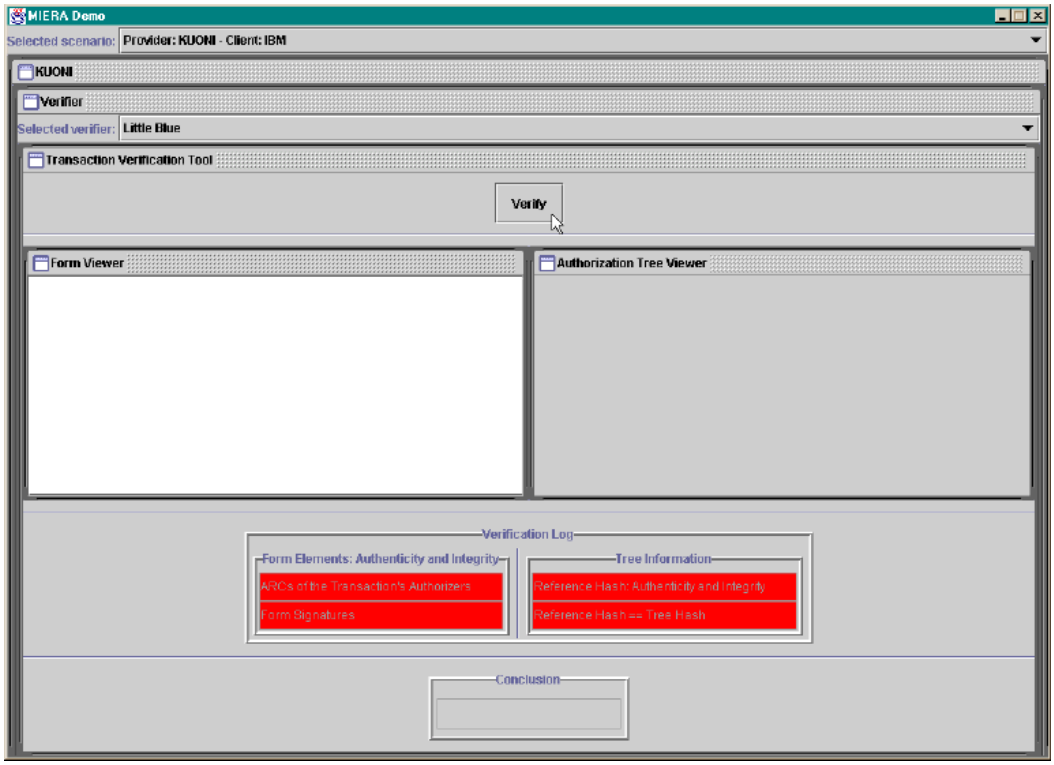

Figure 3.13: Verifier GUI: Verifying the Transaction

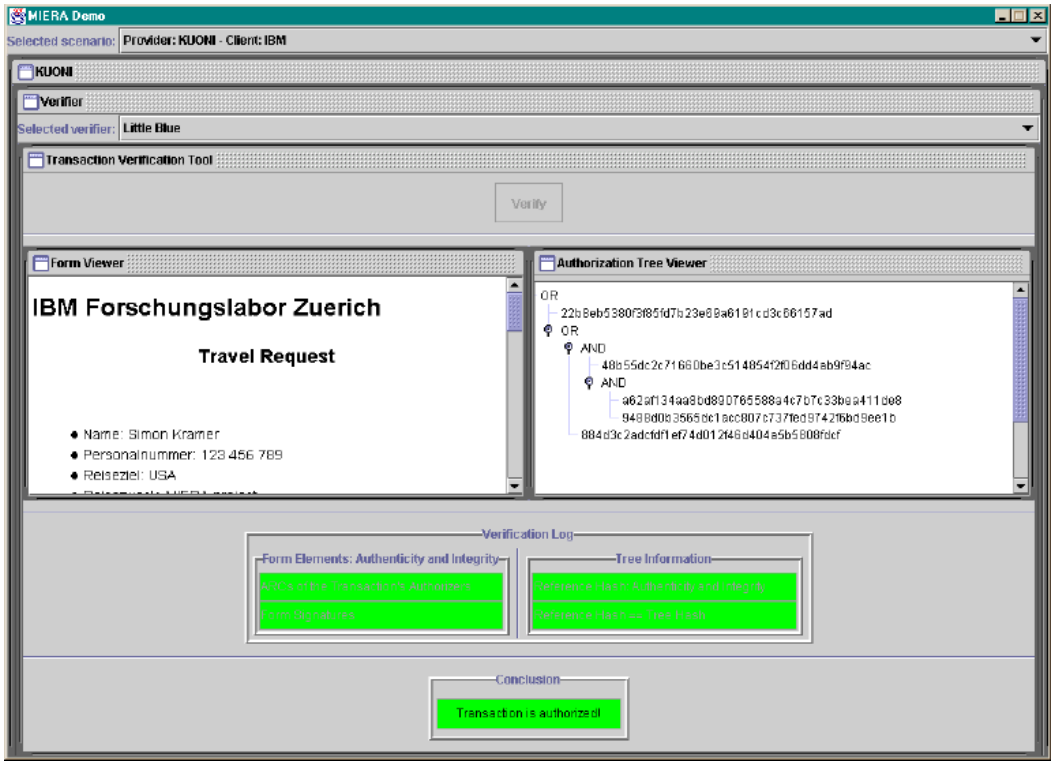

Figure 3.14: Verifier GUI: Verification Result

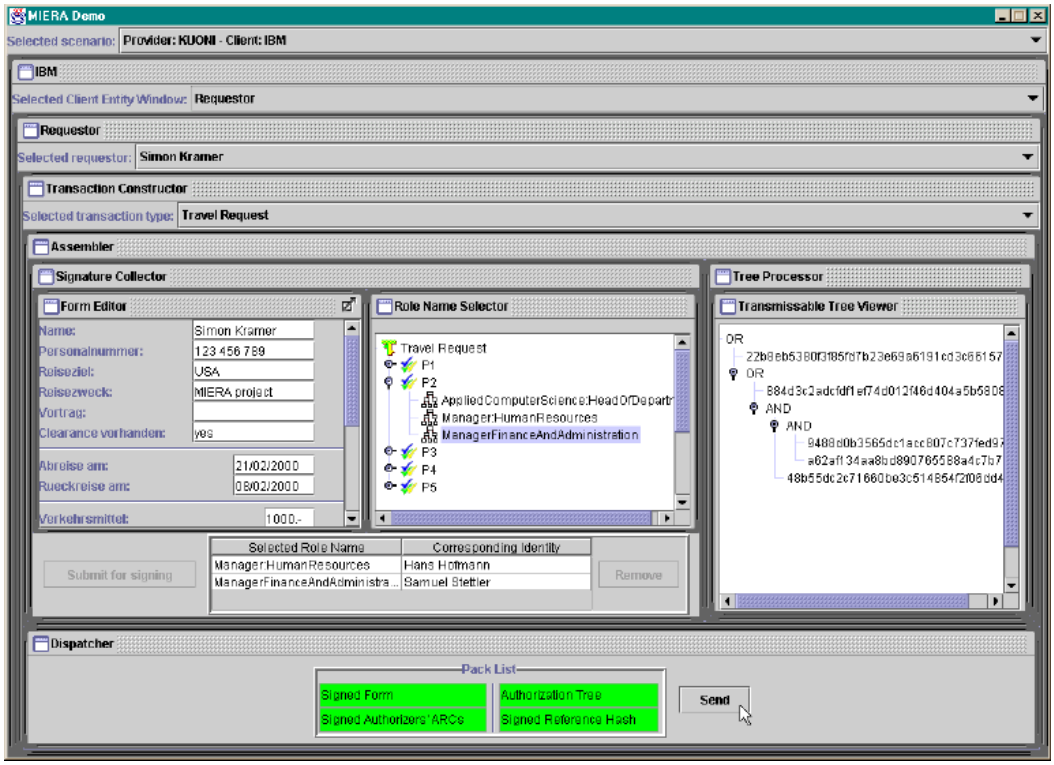

Figure 3.15: A Case of Cheating

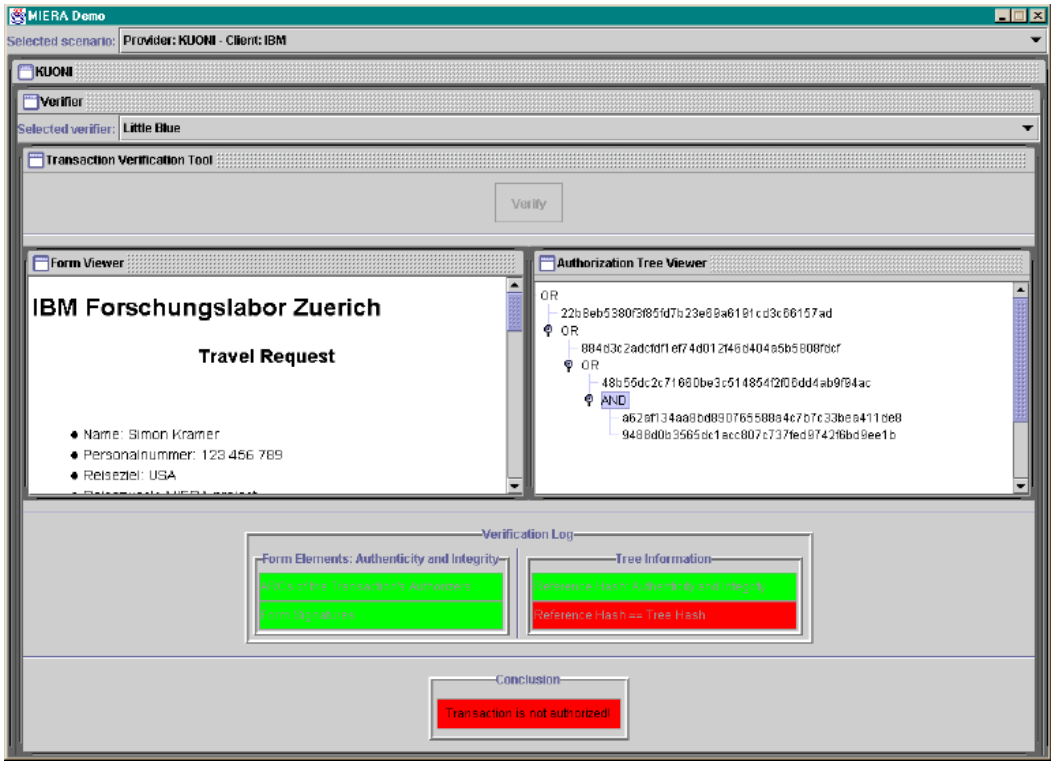

Figure 3.16: Authorisation Failure

Chapter 4

Conclusions

4.1 Discussion

We have given a formal model for MIERA+, an enhancement of the original MIERA scheme as suggested in [1], and provided a detailed documentation of the current implementation.

The formal approach has given us insight and a deeper understanding of the problem. As a result, we are able to distinguish the cases where hashing is necessary in order to reject a bad authorisation tree and where it is not.

4.2 Improvements

4.2.1 MIERA core

As explained in Section 1.3.2.2, the authorisation structure is static and consequently there exists the possibility of its being completely disclosed. Yet since the role certificates are anonymous, the verifier does not gain any knowledge about the real role identities. The only information the verifier gains is about the *relation* between the authorising entities, i.e. the roles[1].

We could remedy this situation if we introduce the concept of *fake anonymous role certificates*. A fake anonymous role certificate is an anonymous role certificate that does not have any corresponding owner. A fake anonymous role certificate could be implemented by assigning a randomly generated value to its name field.

The transaction authority would then include some names of a (fixed) set of fake anonymous role certificates in each permission set of a given authorisation tree, thus introducing redundancy in it. The 'real' and, by definition, necessary and sufficient authorisation structure appears now embedded in a larger than necessary authorisation tree. This new tree still is static but the verifier would never now which part of it does correspond to the real authorisation tree[2].

[1] if the security of the algorithm is not to rely on its secrecy.

[2] In fact, it is not even necessary to actually *use* the fake anonymous role certificates. It is sufficient just to *declare* their use. Since a verifier cannot distinguish between fake and 'real' anonymous role certificates, he can never know whether or not they have been used in a certain authorisation tree.

Another way to prevent the verifier from knowing the authorisation structure could be to use *group signatures* on the permission sets. Yet this would complicate key management.

4.2.2 Demonstrator

The most important enhancement for the demonstrator would be to change the class `TransmittableTreeViewer` into a class allowing modification of the transmittable authorisation tree. This is to allow the demonstration of cheating that nevertheless results in a valid target tree, as described in Section 2.2.4.

In order to make the demonstrator truly portable there should be built an abstraction class for path names. The present implementation runs well on NT machines but causes problems on UNIX machines because their respective file systems do not use the same path name format.

It would be convenient to make the class `DemoModel` serialisable to be able to back-up particular instances of it.

Finally, although the Java Swing package provides a means for displaying HTML forms, it seems however not possible to *access* information once it has been entered into a form. The 'HTML form' for travel requests in the demonstrator has in fact been hard coded.

4.2.3 Integration of MIERA

The MIERA core relies on certificate management, a signature collecting mechanism and a role resolution mechanism. These services would have to be provided in order to use MIERA in a real world context.

Bibliography

[1] H. Ludwig and L. O'Connor. *MIERA: A Method for Inter-Enterprise Role-Based Authorization*, IBM Research, Zurich Research Laboratory, Switzerland.

[2] O. Biberstein. *CO-OPN/2: An Object-Oriented Formalism for the Specification of Concurrent Systems*, Thèse no. 2919, Département d'informatique, Univérsité de Genève, Switzerland, 1997.

[3] G. di Marzo Serugendo. *Stepwise Refinement of Formal Specifications based on Logical Formulae: from CO-OPN/2 Specifications to Java Programs*, Thèse no. 1931, Département d'informatique, École Polytechnique Fédérale de Lausanne, Switzerland, 1999.

[4] J. Zahnd. *Logique élémentaire: Cours de base pour informaticiens*, Presses polytechniques et universitaires romandes, 1998.