ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institute of Theoretical Computer Science

Diploma Thesis

by
Thomas Briner
briner@computerscience.ch

Compiler for Zero-Knowledge Proof-of-Knowledge Protocols

March 2004

Diploma Professor:
Prof. Dr. Ueli Maurer
Institute of Theoretical Computer Science, ETH Zurich
maurer@inf.ethz.ch

Supervisors:
Dr. Jan Camenisch, Endre Bangerter
IBM Research, Zurich Research Laboratory
{jca,eba}@zurich.ibm.com

Abstract

Zero-Knowledge Proof-of-Knowledge protocols are of particular interest for authentication systems as developed for example in the IBM Zurich Research Laboratory in Rüschlikon. There is an arbitrary number of protocol instances that vary in terms of protocol structure, additional restrictions on the preimages of the homomorphisms, as well as regarding the homomorphisms and groups themselves that are used. Depending on the concrete instance these protocols have certain properties that might be useful for such systems.

The generation of a complete protocol instance for specification or testing purposes is a very time-consuming and highly error-prone piece of work. Therefore the purpose of the compiler developed during this diploma thesis was to automate this process.

To this end a compiler input language was created that allows instances of a certain protocol type to be specified and additional types of checks to be added using some auxiliary parameters. The user can choose between different levels of abstraction in specifying a certain protocol instance.

The compiler itself is written in java and is based on traditional object-oriented compiler design patterns. In its library it contains the basic skeleton of the well-known Σ protocol and of the 2Σ protocol developed in Rüschlikon.

The compiler reads the input files with the protocol specifications written in the compiler input language mentioned above and checks for syntactical correctness. Furthermore, some semantic checks on the proper use of the protocol parameters are performed. From this information the compiler generates the protocol instance, written either as LaTeX code or as java source code. The LaTeX code shows the detailed specification of the protocol instance, consisting of the documentation of the algebraic elements involved, the facts that can be deduced if the proof is accepted as well as all steps performed during protocol execution. If java is chosen as target language, it produces runnable java source code. This code is based on an interface hierarchy which models the algebraic constructs that was also developed in the course of this diploma thesis. At runtime, the protocol instance has to be instantiated with concrete implementations, and can then for example be used for testing purposes.

Zusammenfassung

Zero-Knowledge Proof-of-Knowledge Protokolle sind für Authentisierungssysteme, wie sie beispielsweise im IBM Zurich Research Laboratory in Rüschlikon entwickelt werden, von besonderem Interesse. Es gibt eine beliebig grosse Menge von solchen Protokollinstanzen, die im Bezug auf die Protokollstruktur, die zusätzlichen Anforderungen an die geheimen Urbilder der Homomorphismen, aber auch bezüglich der eingesetzten Homomorphismen und Gruppen variieren. Je nach der konkreten Ausprägung haben solche Protokollinstanzen entsprechende Eigenschaften, die für solche Systeme benutzt werden können.

Die Erzeugung des vollständigen Protokollablaufs zwecks Spezifikation oder um eine bestimmte Protokollinstanz zu testen ist eine sehr aufwendige und fehleranfällige Tätigkeit. Aus diesem Grund wurde dieser Vorgang mittels eines Compilers im Rahmen dieser Diplomarbeit automatisiert.

Zu diesem Zweck wurde eine Eingabesprache entwickelt, die es erlaubt eine Instanz eines Protokolltypes zu spezifizieren und mittels verschiedenen Parametern mit zusätzlichen Elementen wie zum Beispiel verschiedenen Arten von Kontrollabfragen auszustatten. Es stehen für diese Spezifikation der Protokollinstanz verschiedene Abstraktions- bzw. Detaillierungsgrade zur Verfügung.

Der Compiler wurde in Java geschrieben und basiert auf dem klassischen objekt-orientiertem Compilerdesign. Er enthält als Protokollbibliothek die Grundstrukturen des bekannten Σ Protokolls und des 2Σ Protokolls, das in Rüschlikon entwickelt wurde.

Der entwickelte Compiler liest Dateien mit Protokollspezifikationen in der erwähnten Inputsprache ein und kontrolliert sie auf syntaktische Korrektheit. Darüber hinaus werden die Protokollparameter auch im Bezug auf die semantisch korrekte Anwendung überprüft. Aus diesen Informationen generiert der Compiler nun die jeweilige Protokollinstanz entweder als LaTeX Code oder als Java source Code. Der LaTeX Code stellt eine detaillierte Spezifikation der Protokollinstanz dar, bestehend aus der Dokumentation der verwendeten algebraischen Elemente, den daraus resultierenden Erkenntnissen im Falle eines akzeptierten Beweises und dem detaillierten Protokollablauf. Die zweite Ausgabeform besteht in lauffähigen Java Source Code. Dieser basiert auf einer abstrakten Interfacehierarchie, die ebenfalls in dieser Diplomarbeit entwickelt wurde, welche die algebraischen Konstrukte modelliert. Zur Laufzeit wird die Protokollinstanz dann mit den konkreten Implementationen instanziert und kann so beispielsweise für verschiedene Arten von Tests verwendet werden.

Contents

List of Figures

Acknowledgements

Many thanks to my supervisors Endre and Jan for their support, their patience and all the interesting discussions and the encouragement during my diploma thesis.

Thanks to Professor Maurer for his challenging lectures on the fascinating topic of crytography which I enjoyed very much and to his assistants.

Many thanks to Herrn Dubach for his support and advice in all questions that could arise during these years at ETH.

Thanks to Daniel Engeler, the expert for all LaTeX problems, Daniel Lutz for his linux support and my roommates in C231 at the Research Lab.

1 Introduction

In this section I will give a brief overview of the concept of zero-knowledge proof-of-knowledge protocols, and sketch some of the main properties of such protocols. In the second part of the introduction, I will describe the motivation for developing such a compiler and the main goals of the project.

1.1 Zero-Knowledge Proof-of-Knowledge Protocols

A zero-knowledge proof-of-knowledge protocol is a special instance of the class of interactive proofs. Two entities are involved: One that proofs a certain fact, called the prover, and the other that verifies whether the proof is correct and will be accepted, called the verifier. These entities perform different types of computations and exchange messages. Such a protocol consists of all the information about the computations, message exchanges and decisions that have to be made.

In the case of a zero-knowledge proof-of-knowledge protocol, the prover wants to prove a certain knowledge. This knowledge will be called the secret. The secret may consist of several different parts. For our types of zero-knowledge proof-of-knowledge protocols, these secrets always are preimages of homomorphic functions (or simply, homomorphisms).

A function $\zeta : G \to H, x \mapsto \zeta(x)$ is called homomorphic if

$$\zeta(x_1) \otimes \zeta(x_2) = \zeta(x_1 \oplus x_2),$$

where $x_1, x_2 \in G$ and $\otimes$ is the operation in group H, $\oplus$ the operation in group G. Each group can consist of a combination of subgroups e.g. $G : G_1 \times G_2 \times \mathbb{Z} \times \mathbb{Z}_m$. The prover and the verifier both know the images of the secrets, the homomorphisms and some additional information. The preimages, in contrast, are known only to the prover.

A zero-knowledge proof-of-knowledge protocol can have different properties. I will sketch three of them, which are important in the context of this diploma thesis, in an informal way.

Completeness or Validity A zero-knowledge proof-of-knowledge protocol is called complete or valid if a prover who knows the secret and acts correctly according to the protocol can be sure that the proof will be accepted, provided the verifier is honest.

Soundness A zero-knowledge proof-of-knowledge protocol has the soundness property if a proof will be accepted if and only if the secret is known to the prover. No prover is able to convince a verifier in such a way that the proof ends up in the accepting state without knowing the secret.[1] This property is proved by the definition of an appropriate knowledge extractor.

Zero-Knowledge Property The zero-knowledge property guarantees that the verifier will not obtain any information about the secret in addition to those he or she already had before the proof. No information will be given away by the prover. This property is proved by the existence of an appropriate simulator, and it might depend on a security parameter.

[1] This property has to hold with an overwhelming probability.

The protocols generated by this compiler will all have the completeness and the zero-knowledge property. This is what is meant by saying that a protocol is correct. Whether the protocol also has the soundness property depends on the actual protocol specified by the user.

1.2 Motivation and Goal of the Diploma Thesis

At the IBM Research Laboratory in Rüschlikon, an anonymous credential system called idemix (identity mixer) was designed that uses zero-knowledge proof-of-knowledge protocols for authentication. This system and successor projects that might be developed will need many different types of such zero-knowledge proof-of-knowledge protocols with different properties.

A protocol instance depends to a large extent on many different parameters that may be chosen independently. The choice of the protocol parameters can influence the overall structure of the protocol instances as well as each single operation. The secret structure of such a protocol that can be used for real credential systems can be very complex as shown by the example in section B of the appendix which is a part of one of the prototype protocols as used in idemix. The size of a specification that describes all actions performed and all operations on all variables is huge. For all these reasons the specification or implementation of such a protocol instance is a very time-consuming and error-prone piece of work. Therefore this process should be automated.

The goal of this diploma thesis is it to develop a compiler that takes all parameters as input to specify a certain instance of a zero-knowledge proof-of-knowledge protocol and generates a specification or an implementation of this protocol instance from this input. The specification that shows the execution of the protocol is written as LaTeX code. The implementation should be written in executable java source code.

The compiler should deal with two main types of protocols, the well-known Σ protocol and the 2Σ protocol developed in the Research Laboratory in Rüschlikon. It should provide support for different levels of abstraction in specifying the protocol instances and its components. Beyond the specification generated in LaTeX code or the generation in java source code, additional information about the properties of a protocol instance would be of interest, such as they are defined in section 1.1.

As a first step a compiler input language that covers the range of all these protocol instances has to be developed. Based on this language the compiler is written that translates the protocol instance into one of the two target languages. For the LaTeX code generated the main focus is on comprehensibility and clarity whereas in generating java source code the emphasis lies on generality and efficiency.

2 Generalized Protocol

All these different kinds of protocol types with their instances depending on the specification of the desired protocol can be considered as instances of one general protocol who covers the whole range of protocol instances that can be generated by this compiler. I will describe first the two main types covered by the generalized protocol and then go on with several variations of the protocol instances.

2.1 Σ Protocol

The Σ protocol is a generalization of well-known concrete zero-knowledge proof-of-knowledge protocols such as the Schnorr Identification Protocol[7] or the Guillou-Quisquater Identification Scheme[5]. This generalization was introduced by Ronald Cramer and Ivan Damgård and is described in [3]. It is a protocol using three message rounds as shown in figure 2.1.[2]

Now I will now have a look at the properties of such a protocol as introduced in section 1.1. The completeness property is given simply using the homomorphic property. In order to decide whether the soundness property is given one has to answer the question, whether a knowledge extracter can be specified for the homomorphisms used in the protocol. The first step is to find a collision which occurs if two different challenges c_1, c_2 for the same first message can be answered each with the correct corresponding third message containing the variable s_1, s_2. That gives the following collision equation:

$$\zeta(x)^{c_2-c_1} = \frac{\zeta(s_1)}{\zeta(s_2)}.$$

In order to specify a knowledge-extractor one has to be able to compute a value of the preimage x from this collision equation. This can be done using the extended euclid algorithm but it needs additional knowledge about the group. One needs to know an integer t and a group element u, such that

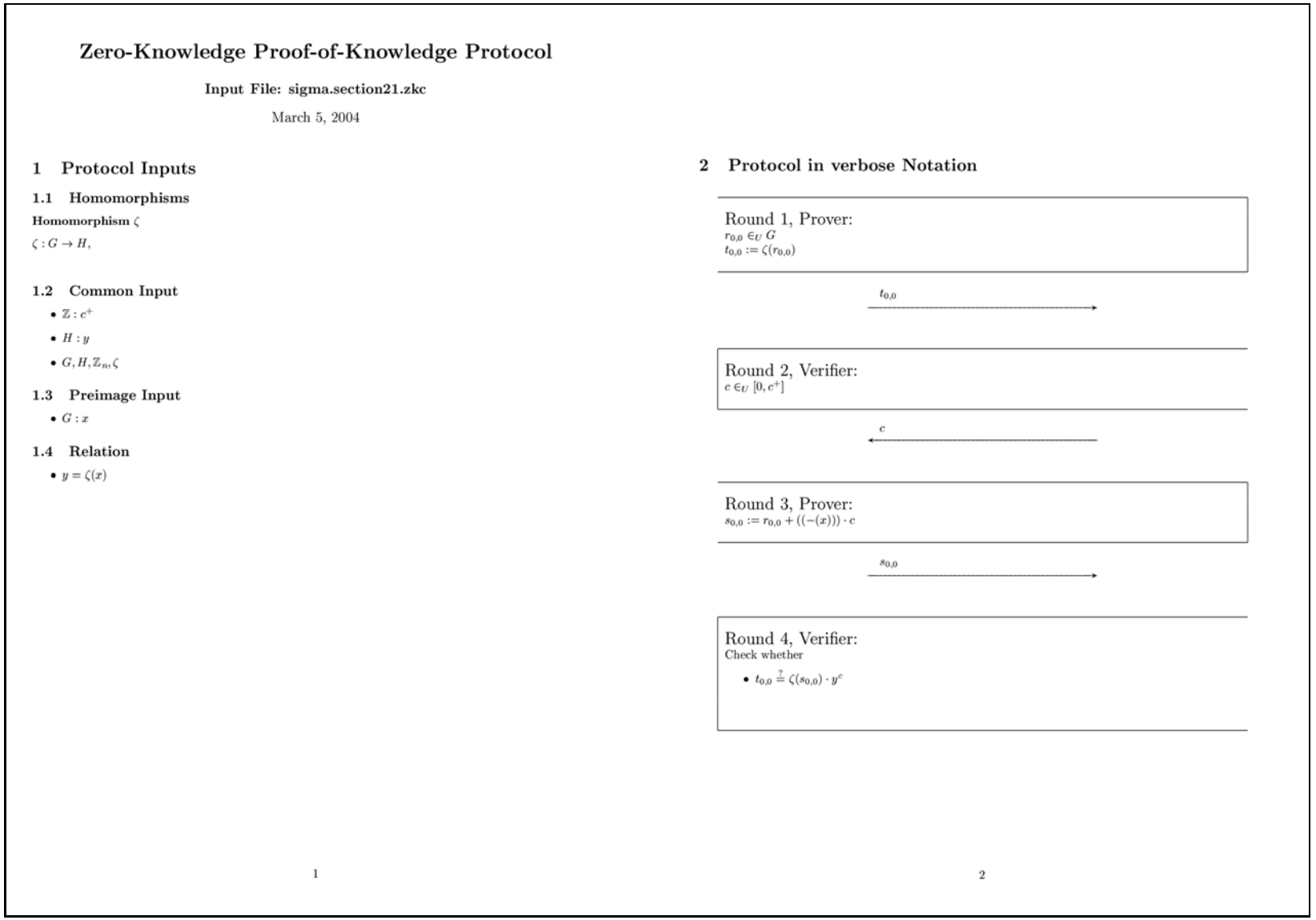
$$x^t = u.$$

If the group order is known, the group order can be used as value for t and the neutral element of the group for the element u. In this case a knowledge extractor can be specified and the soundness property is fulfilled. If the group order is unknown, the knowledge extractor can not be "built" that way and the soundness property using the Σ Protocol is not given.

The zero-knowledge property depends on the existence of a simulation algorithm, such that a set of tuples (s_i, c_i, t_i) created from the simulator is indistinguishable from the tuples generated from a real proof. As these tuples can be produced for the Σ Protocol without the knowledge of the secret, the protocol fulfills the zero-knowledge property.

The Σ Protocol is defined for a general homomorphism. It might be instantiated with an arbitrary homomorphism as it uses only the basic homomorphic property as definied in section 1.1. The overall structure of the protocol remains the same but depending on the homomorphism and for example the structure of its domain an co-domain some things might change. I will later illustrate these changes for some examples of homomorphism structures showing the Σ protocol for those special cases.

[2] All pictures of protocols were generated using the compiler developed in this diploma thesis.



Zero-Knowledge Proof-of-Knowledge Protocol

Input File: sigma.section21.zkc

March 5, 2004

1 Protocol Inputs

1.1 Homomorphisms

Homomorphism ζ

$\zeta : G \to H,$

1.2 Common Input

- $\mathbb{Z} : c^+$
- $H : y$
- $G, H, \mathbb{Z}_n, \zeta$

1.3 Preimage Input

- $G : x$

1.4 Relation

- $y = \zeta(x)$

2 Protocol in verbose Notation

Round 1, Prover:
$r_{0,0} \in_U G$
$t_{0,0} := \zeta(r_{0,0})$

$t_{0,0}$ →

Round 2, Verifier:
$c \in_U [0, c^+]$

← c

Round 3, Prover:
$s_{0,0} := r_{0,0} + ((-(x))) \cdot c$

$s_{0,0}$ →

Round 4, Verifier:
Check whether

- $t_{0,0} \stackrel{?}{=} \zeta(s_{0,0}) \cdot y^c$

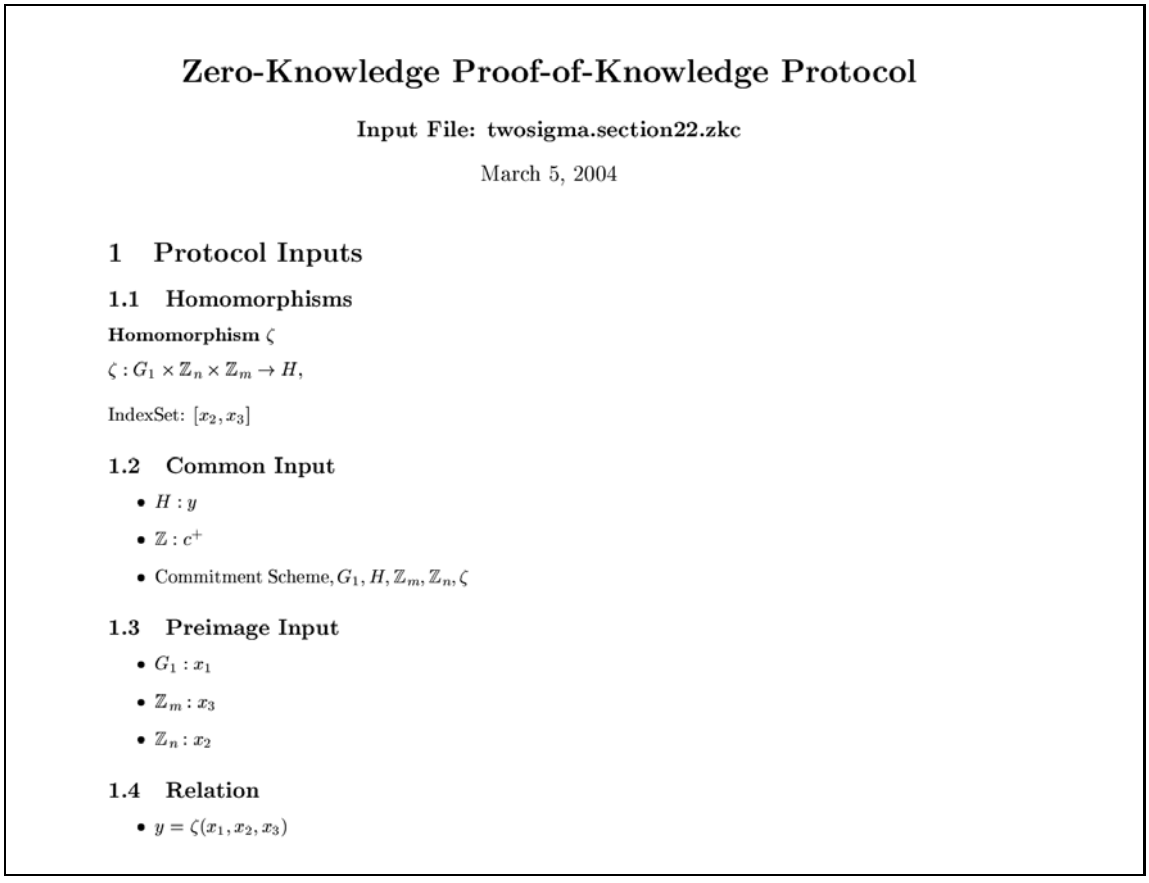
Figure 2.1: Basic Structure of the Σ Protocol

2.2 2Σ Protocol

The 2Σ protocol is an extension of the Σ protocol protocol. It can be used for those cases where the soundness property is not given using the Σ protocol protocol. For all those groups in the domain where a knowledge extractor for the Σ protocol protocol does not work, the 2Σ protocol can be applied. The result is that according to the strong RSA problem, a knowledge extractor can be specified even if the group order is unknown.

An index set has to be specified that decides on which preimages an additional homomorphism should be created. The 2Σ protocol needs additional primitives such as an instance of the strong RSA problem that has to be generated and a commitment scheme.

If the 2Σ protocol is chosen, but the index set that specifies for which preimages the additional homomorphism should be applied, is empty, this corresponds exactly to the Σ Protocol. As the 2Σ protocol is an extension of the Σ protocol , all adaptations that were mentioned in section 2.1 influence the 2Σ protocol too. Figures 2.2 and 2.3 shows the basic structure of the 2Σ protocol:



Zero-Knowledge Proof-of-Knowledge Protocol

Input File: twosigma.section22.zkc

March 5, 2004

1 Protocol Inputs

1.1 Homomorphisms

Homomorphism ζ

$\zeta : G_1 \times \mathbb{Z}_n \times \mathbb{Z}_m \rightarrow H,$

IndexSet: $[x_2, x_3]$

1.2 Common Input

- $H : y$
- $\mathbb{Z} : c^+$
- Commitment Scheme, $G_1, H, \mathbb{Z}_m, \mathbb{Z}_n, \zeta$

1.3 Preimage Input

- $G_1 : x_1$
- $\mathbb{Z}_m : x_3$
- $\mathbb{Z}_n : x_2$

1.4 Relation

- $y = \zeta(x_1, x_2, x_3)$

Figure 2.2: Basic Structure of the 2Σ Protocol, Input Section

Figure 2.3: Basic Structure of the 2Σ Protocol, Protocol Execution

2.3 Access Structure

Up to now I have been speaking about "the secret" as though there could be only one. In this case the prover either knows the secret or he does not. But a zero-knowledge proof-of-knowledge protocol can be used not only for this simple case but the secret may consist of a set of preimages connected through a certain structure called the access structure.

The access structure can be written as a boolean formula where the homomorphisms with their corresponding preimages are the litterals. If the prover knows a subset of these preimages and those positions are set to true in the boolean formula and the unknown ones to false, the formula should evalutate to true in order to make sure that the prover will accept the proof.[3] The access structure may be an arbitrary boolean formula, as long as there is a secret sharing scheme that fits with this structure.

Each boolean formula can be written in disjunctive normal form (DNF). For the access structure in the DNF, each clause corresponds to a set that makes the formula evaluate to true, which will be called a qualified set. Therefore this is a subset of all preimages that will make an honest verifier accept the proof. If the prover knows at least one of the qualified sets and if he acts correctly, the proof should be accepted.

In order to convince the verifier, the prover will follow the proof as usual for those preimages he really knows. For the other ones he will use the simulation algorithm that is used in order to proof the zero-knowledge property. As the distribution of tuples from these two sources are indistiguishable, the verifier will not get any information about which set of preimages the prover does know and which one he does not, as long as it fulfills the requirements of the access structure. The whole protocol will be processed over the homomorphisms in parallel. Therefore we will call this type of protocol structure the "parallel protocol".

The challenge that the verifier chooses has to be split up into several pieces, called shares, that are consistent according to the secret sharing scheme. We have chosen to generate a share for each homomorphism that the user gives as input[4]. If they belong to the same clause in the DNF what means that they are in the same qualified set, this would not be necessary. It would be possible to choose one share per qualified set. This solution would have a huge disadvantage. If we create a share for each qualified set, the number of shares that could be needed would explode. In the solution where every homomorphism has his own share, the upper bound for the number of shares is simply the number of homomorphisms.

In order to work with this set of shares, we need a secret sharing scheme that provides two algorithms:

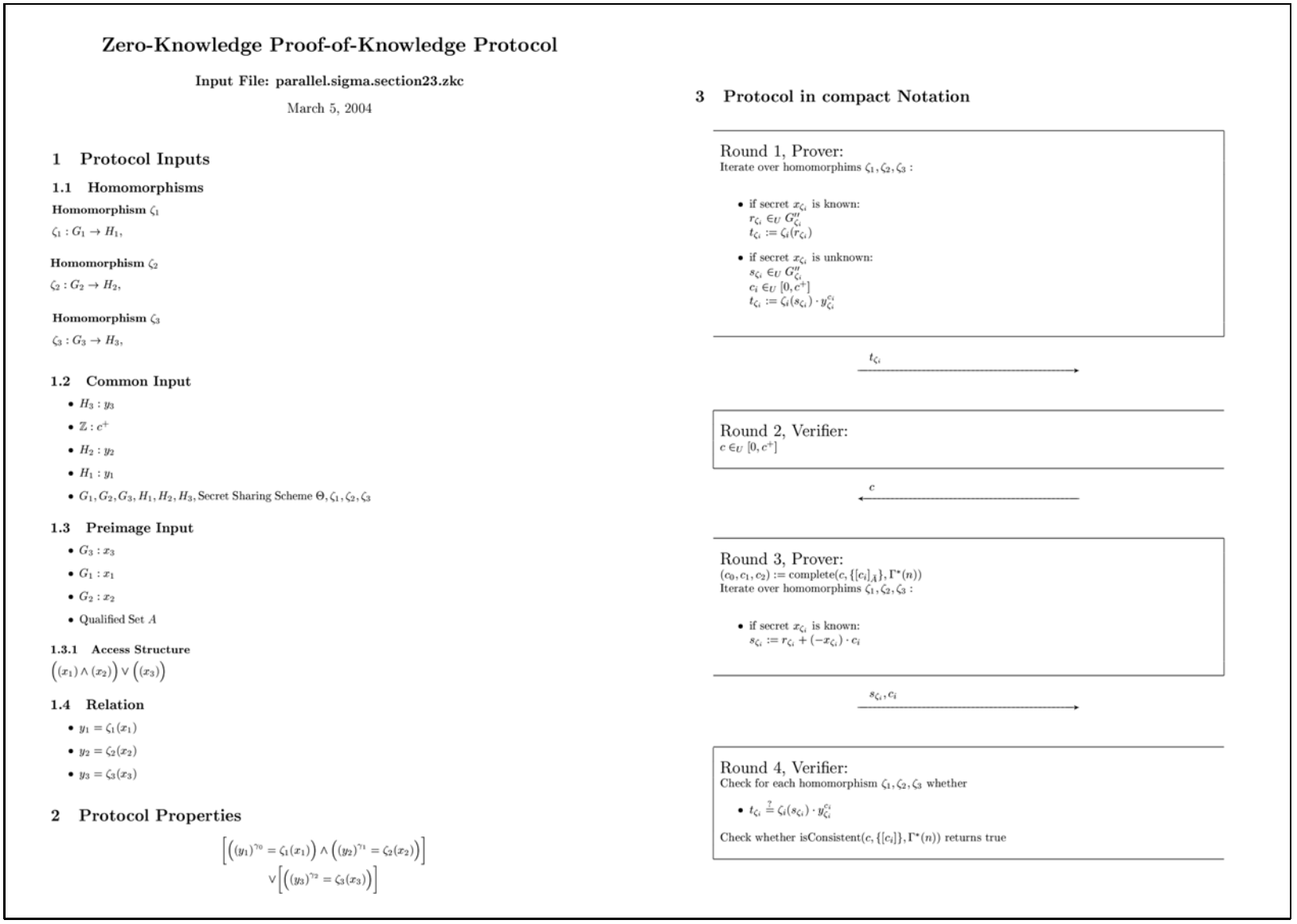
- An algorithm that will be called "complete" that takes as input the challenge, a set of shares and the access structure. If the set of shares given as input can be supplemented with one of the qualified sets and the result corresponds to the set of all shares, the algorithm will return such a complete set of shares that is consistent with the challenge chosen by the verifier.

[3] Provided the prover acts correctly and the verifier is honest.

[4] This principle has to be broken in case of constraints on the preimages as explained in section 2.5.2.

- An algorithm called "isConsistent" that takes as input the challenge, the complete set of shares and the access structure. It checks whether the input is a consistent set according to the challenge and the access structure. As output it returns the corresponding boolean value.

Figure 2.4 shows an example of a parallel Σ protocol with several homomorphism and a given access structure.



Zero-Knowledge Proof-of-Knowledge Protocol

Input File: parallel.sigma.section23.zkc

March 5, 2004

1 Protocol Inputs

1.1 Homomorphisms

Homomorphism ζ_1

$\zeta_1 : G_1 \to H_1,$

Homomorphism ζ_2

$\zeta_2 : G_2 \to H_2,$

Homomorphism ζ_3

$\zeta_3 : G_3 \to H_3,$

1.2 Common Input

- $H_3 : y_3$
- $\mathbb{Z} : c^+$
- $H_2 : y_2$
- $H_1 : y_1$
- $G_1, G_2, G_3, H_1, H_2, H_3$, Secret Sharing Scheme $\Theta, \zeta_1, \zeta_2, \zeta_3$

1.3 Preimage Input

- $G_3 : x_3$
- $G_1 : x_1$
- $G_2 : x_2$
- Qualified Set A

1.3.1 Access Structure

$\Big((x_1) \wedge (x_2)\Big) \vee \Big((x_3)\Big)$

1.4 Relation

- $y_1 = \zeta_1(x_1)$
- $y_2 = \zeta_2(x_2)$
- $y_3 = \zeta_3(x_3)$

2 Protocol Properties

$$\Big[\Big((y_1)^{\gamma_0} = \zeta_1(x_1)\Big) \wedge \Big((y_2)^{\gamma_1} = \zeta_2(x_2)\Big)\Big] \vee \Big[\Big((y_3)^{\gamma_2} = \zeta_3(x_3)\Big)\Big]$$

3 Protocol in compact Notation

Round 1, Prover:
Iterate over homomorphims $\zeta_1, \zeta_2, \zeta_3$:

- if secret x_{ζ_i} is known:
  $r_{\zeta_i} \in_U G''_{\zeta_i}$
  $t_{\zeta_i} := \zeta_i(r_{\zeta_i})$
- if secret x_{ζ_i} is unknown:
  $s_{\zeta_i} \in_U G''_{\zeta_i}$
  $c_i \in_U [0, c^+]$
  $t_{\zeta_i} := \zeta_i(s_{\zeta_i}) \cdot y_{\zeta_i}^{c_i}$

t_{ζ_i} →

Round 2, Verifier:
$c \in_U [0, c^+]$

← c

Round 3, Prover:
$(c_0, c_1, c_2) := \text{complete}(c, \{[c_i]_{\tilde{A}}\}, \Gamma^*(n))$
Iterate over homomorphims $\zeta_1, \zeta_2, \zeta_3$:

- if secret x_{ζ_i} is known:
  $s_{\zeta_i} := r_{\zeta_i} + (-x_{\zeta_i}) \cdot c_i$

s_{ζ_i}, c_i →

Round 4, Verifier:
Check for each homomorphism $\zeta_1, \zeta_2, \zeta_3$ whether

- $t_{\zeta_i} \stackrel{?}{=} \zeta_i(s_{\zeta_i}) \cdot y_{\zeta_i}^{c_i}$

Check whether isConsistent$(c, \{[c_i]\}, \Gamma^*(n))$ returns true

Figure 2.4: Example of a parallel Σ Protocol

2.4 Homomorphism Structure

A homomorphism maps from a tuple of groups called the domain to another tuple of groups called the co-domain. The generated protocol instance depends strongly on the composition of these tuples.

2.4.1 Homomorphism with Compound Groups in the Domain or Co-Domain

If a tuple of groups of either the domain or the co-domain consists of more than one single group, the whole tuple can be considered as a compound group. In case of compound groups, each group operation has to be processed as group operations in the subgroups of the domain and the co-domain.

Figure 2.5 is an example of a Σ protocol with a homomorphism consisting of compound groups.

2.4.2 Working with infinite groups

Each group consists of a certain number of group elements. Groups with an infinite number of elements are called infinite groups. If infinite groups are involved in the domain of a homomorphism, an integer interval needs to be specified for each preimage belonging to an infinite group. The reason for this is that no random choice from an infinite number of elements is possible, that has to be uniformly distributed. The probability for choosing each single element would be zero. Therefore the the random elements are chosen out of a certain range. In order to guarantee the zero-knowledge property as defined in section 1.1 an additional security parameter for the mapping of the range where the secret x is chosen from to the range where the variables r and s are chosen from is required.

For performance reasons, the average value of this integer interval will be used as a kind of shift in order to make computations easier an less time consuming. Therefore the computations have to be adapted for all elements that belong to an infinite group.

Figure 2.6 is an example of a Σ protocol where the domain contains infinite groups.

2.5 Additional Options

There are some further options that can be chosen by the user. They provide additional features that might be necessary in order to achieve some properties. These options have been introduced because of the structure of protoype protocols that were useful for systems mentioned in section 1.2.

2.5.1 Interval checks

The option of using interval checks can be used as an additional feature if infinite groups are involved in this protocol. In order to know whether the preimages and the value r could have been chosen from the appropriate interval, the verifier can check whether or not s lies within the expected interval.[5] If it is not, the prover did missbehave in some way and the prove will be rejected.

[5]The interval where the preimage should come from is known as common input. From this interval together with the challenge range and a security parameter, the interval in which the temporary variable s should be can be computed.

Figure 2.5: Instance of the Σ Protocol with Compound Groups in the Domain and Co-Domain.

Figure 2.6: Example of a Σ protocol with infinite groups in the domain.

2.5.2 Constraints on preimages

Preimages of the same group may be connected by some constraints of the form

```
(x_1 = 2*x_4 + x_6) && (x_3 = 6*x_5).
```

These constraints are taken into account each time when an element is chosen randomly, e.g. the temporary value r corresponding to the preimages that are involved in such constraints. This random choice will be performed in the following way:

- Each variable that is never on the left hand side in none of these constraints is chosen randomly as usual.
- Each variable that is on the left side of a constraint is not chosen randomly but is computed according to the depencies stated in the constraint.

At the end of the protocol when the verifier chooses whether to accept the protocol or not, he will now check, whether the constraints still hold on the s values corresponding to the preimages that were used in the constraints in addition to the other checks he performs.

These constraints have an influence on the protocol execution much stronger than it seems to at first glance. According to these constraints, the computations in different homomorphism depend on each other. Therefore the principle that each homomorphism is treated seperately with a share of its own has to be broken. The homomorphisms that are connected through constraints are combined. The new construct that might consist of several user defined homomorphism is called an atom. If constraints are involved, the processing unit on the execution changes from the user defined homomomorphisms to the atoms. The rules for assembling these atoms are described in section 3.3.2 in detail.

Figures 2.7 and 2.8 show a protocol with additional interval checks and constraints on the preimages.

Zero-Knowledge Proof-of-Knowledge Protocol

Input File: additionalOptions.zkc

March 5, 2004

1 Protocol Inputs

1.1 Homomorphisms as defined in Input File

Homomorphism ζ_1

$\zeta_1 : \mathbb{Z} \to H$,

$G'_{\zeta_1} = [a, b]$
$G''_{\zeta_1} = [-2^{\sigma_r} c^+ [\frac{b-a}{2}], 2^{\sigma_r} c^+ [\frac{b-a}{2}]]$
$\zeta'_1 = \zeta_1 | G'_{\zeta_1}$
$\zeta''_1 = \zeta_1 | G''_{\zeta_1}$

Homomorphism ζ_2

$\zeta_2 : \mathbb{Z}^2 \to H$,

$G'_{\zeta_2} = [a, b]^2$
$G''_{\zeta_2} = [-2^{\sigma_r} c^+ [\frac{b-a}{2}], 2^{\sigma_r} c^+ [\frac{b-a}{2}]]^2$
$\zeta'_2 = \zeta_2 | G'_{\zeta_2}$
$\zeta''_2 = \zeta_2 | G''_{\zeta_2}$

1.2 Homomorphisms as used in Protocol

- $\psi_0 = \zeta_1 \times \zeta_2$

1.3 Common Input

- $H : y_1, y_2$
- $\mathbb{Z} : a, b, c^+$
- H,Secret Sharing Scheme Θ, ζ_1, ζ_2

1.4 Preimage Input

- $\mathbb{Z} : x_1, x_2, x_3$
- Qualified Set A

1.4.1 Access Structure

$\Big((x_1) \wedge (x_2, x_3)\Big)$

1.4.2 Constraints on Preimages

- $x_1 = 3 \cdot x_2$

1.4.3 IntervalChecks

Interval checks are performed to the variables corresponding to these secret preimages:

- x_1

1.5 Relation

- $y_1 = \zeta_1(x_1)$
- $y_2 = \zeta_2(x_2, x_3)$

2 Protocol Properties

$$\Big[\Big((y_1)^{\gamma_0} = \zeta_1(x_1)\Big) \wedge \Big((y_2)^{\gamma_0} = \zeta_2(x_2, x_3)\Big) \wedge \Big(x_1 = 3 \cdot x_2\Big) \wedge \Big(\sigma_{0,0} \in [-2^{(\sigma_r+2)} c^+ \tfrac{b-a}{2}, 2^{(\sigma_r+2)} c^+ \tfrac{b-a}{2}]\Big)\Big]$$

Figure 2.7: Instance of the Σ Protocol with Constraints on Preimages and Interval Checks, Input Section.

3 Protocol in verbose Notation

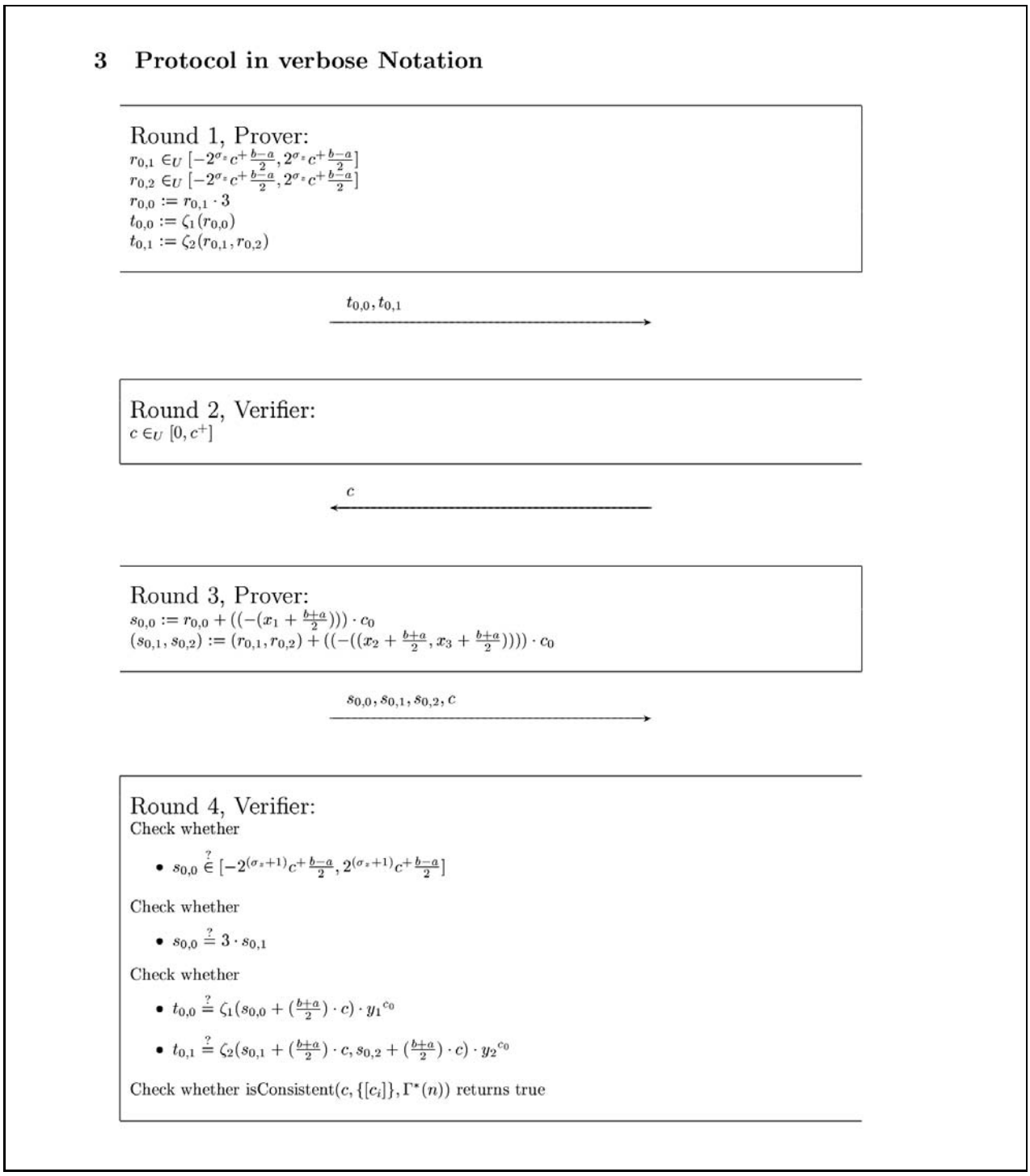


Figure 2.8: Instance of the Σ Protocol with Constraints on Preimages and Interval Checks, Protocol Execution.

3 Input Language

3.1 Overview of an Input File

The input file consists of two main parts. Part 1 is the declaration and definition part. In this part all algebraic elements that are used for the protocol have to be declared and defined such as group elements, groups, homomorphisms. In the second part, the protocol parameters that define the protocol instance which should be generated are chosen.

These parts have to remain always in this order. The transistion from one part to the other does not need any special sign but is recognized simply by the keywords that belong to the different parts.

The input file contains all information about the protocol instance that should be generated. There are no additional configuration files.

The java line comments `//` can be used as comments at any place in the input file in order to add comments. Exactly as in java, all content from the comment sign `//` up to the end of the line will not be considered any further by the compiler.

There are several examples of input files together with the generated output that can be found in the appendix in section C.

3.2 Part 1: The Declarations, Assignments and Definitions

In this first part of the file all algebraic elements that will be used within the protocol instance have to be declared and defined.

3.2.1 Declarations

The Input File starts with the declaration section. Each element that is used later in the file has to be declared beforehand. These elements can be of the following types:

- `Homomorphism`
- `Group`
- `GroupElement`
- `IntegerInterval`
- `IntegerConstant`

The declarations are always of the structure

```
Type variableName;
```

except for the group declaration. Each group needs a graphical symbol for the group operation assigned to it. The symbol is needed in order to write the LaTeX code for the group operation. There are two possibilies, the group operation can be displayed either as a "+" or as a "$\cdot$". The assignment of a sign to a group has no further influence on the protocol. It does not say that this is either an additive or a multiplicative group. These symbols are only needed in order to generate the LaTeX code. If a group G is defined as `Group (G,+)`, an expression could look like that:

$$x_1 + 4 * x_2,$$

where x_1 and x_2 are group elements of G and 4 is an integer.

If the group G is defined as `Group (G,*)`, the same expression would look like that:

$$x_1 \cdot (x_2)^4,$$

The order in which the element declarations appear is arbitrary and does not matter.

```
GroupElement x;
Group (G,+);
Homomorphism zeta;
GroupElement y;
```

is possible as well as any other order.

The infinite additive group $\mathbb{Z}$ does not need to be declared as this a predefined symbol. For this reason it is not allowed to use the letter `Z` as a variable name for any variable.

The finite additive groups play a special role too and therefore another type of variable names has predefined semantics: Any variable name starting with an uppercase `Z` followed by a second letter stands for an additive finite group. The order of the group is specified by the second letter.

Here is an example of such a declaration:

```
Group Zm, Zn;
```

Both groups `Zm` and `Zn` are additive finite groups with the group orders m respectively n. Therefore they do not need the specification of the corresponding group operation sign in the declaration as it would be necessary for other groups.

Choosing Variable Names A variable name according to the `Ident` definition in the EBNF has to start with a letter – uppercase or lowercase. The following symbols can be either letters, numbers or underscore symbols "_".

The choice of variable names has some further restrictions: If a user chooses as name for a group element the variable name `g_1` and all internal names are chosen according to the default setting, the compiler will complain and throw the following exception:

```
Error: Variable name g_1 in conflict with a keyword!
```

The problem is that the compiler creates a lot of temporary variables for the execution of the protocol. These variables have predefined default names that are shown in section 3.3.5. The variable `g_1` is now in conflict with one of these predefined default names, namely with the SRSAGenerator, who's name is "`g`" by default. The compiler will use for the bases of the additional homomorphism for the 2Σ protocol the names `g_0`, `g_1`, ... Therefore this variable name was not accepted.

The rule is that each variable name may be used only once. If there is such a collision of a variable name that is chosen by the user with a predefined variable name, there are two possibilities to fix it:

1. Change the desired variable name of the group element in order to make sure that there is no conflict with one of the predefined variable names.
2. Change the internal naming convention by overwriting the default name for the SRSAGenerator and setting it for example to `p`. Now the compiler will accept the name `g_1` for a group element as there is no longer a conflict with a predefined name.

Notation Details and Shortcuts Here is a list of some notation details and of some shortcuts that were introduced for reasons of convenience.

- Instead of declaring each variable separately

  ```
  GroupElement x;
  GroupElement y;
  GroupElement z;
  ```

  they might be combined into a single declaration:

  ```
  GroupElement x, y, z;
  ```

To combine the elements into one single declaration does not mean anything about the properties of these elements, e.g. the group elements do not have to be members of the same group.

- A variable name has to start with a letter (upper case or lower case) that might be followed by letters, numbers or the underscore symbol _ in arbitrary order.
- The variable names will be displayed in LaTeX in the following way: The string is divided by the first occurence of the underscore symbol "_". The part that follows this symbol will be displayed in LaTeX mode as the index of the variable with the name according to the first part, e.g. `G_1` will be displayed as G_1. If the first part — the variable name — is a letter of the greek alphabet it will be displayed by the according greek letter, e.g. `zeta_1` will be displayed as ζ_1.

  In Java, the names of the variable remain exactly the same as they are declared in the input file.
- It might be convenient to choose variable names with variables with indices. There is a shortcut notation for this case where several variable names that differe only on the indices should be declared. Instead of writing

  ```
  GroupElement x_1, x_2, x_3, x_4, x_5;
  ```

  it can be written in a kind of array notation as

  ```
  GroupElement x_[1..5];
  ```

  Behind the scenes this statement is expanded into the statement with all variables declared separately and is therefore equivalent. This shortcut is only supported throughout part 1 of the input file.

EBNF for the Declaration Section The syntax of the declaration section is described by the following set of productions in EBNF notation:

Declaration ::= GroupDeclaration | (("GroupElement" | "Homomorphism" | "IntegerInterval" | "IntegerConstant") IdentList)";".

GroupDeclaration ::= "Group" "(" Ident [ArrayNotation] "," ("+" | "*") ")" ["," "(" Ident [ArrayNotation] "," ("+" | "*") ")"].

IdentList ::= Ident [ArrayNotation] { "," Ident [ArrayNotation]}

ArrayNotation ::= "[" Number "." "." Number "]".

Ident ::= Letter { Letter | Number | "_" }.

Letter ::= "A" | "B" | ... | "Z" | "a" | ... | "z".

Number ::= "0" | ... | "9" { "0" | ... | "9" }.

3.2.2 Assignments

Up to now group elements and groups do not have any connections. In this section we have to assign each group element to one of the groups that were declared in the first section or to the additive infinte group $\mathbb{Z}$.

Here is an example of these assignments:

```
AssignGroupMember(Zm, x_1);
AssignGroupMember(Z, {x_2, x_3, x_4);
AssignGroupMember(H_1, {y_[1..3]});
AssignGroupMember((i_1 subset Z), x_5);
```

The first assignment assignes the group element x_1 to the group Z_m. If there is a single group element assigned, no brackets are needed. In the second assignments, several group elements are assigned to the same group. Notice the braces that are mandatory as soon as the assignment involves more than one group element. The third assignment has exactly the same semantics as the second one as it assignes several group elements to a single group.

The fourth assignment is a special case. If a user decides to specify a homomorphism as a concrete function as explained in section 3.2.3 there is no way for an element that belongs to the additive infinite group to specify the integer interval it should be chosen from. This is the reason why the integer interval can be specified directly in the assignment. Please notice that each preimage that belongs to the additive infinite group needs to be assigned to exactly one integer interval. This is done in case of abstract homomorphism declaration implicitely over the connection through the homomorphism where the interval is given in the abstract structure. In case of concrete homomorphism definition this has to be done here in the assignment section.

EBNF for the Assignment Section Here is the EBNF syntax for the assignment section:

Assignment ::= "AssignGroupMember" GroupMemberAssignmentList ";".

GroupMemberAssignmentList ::= "(" Ident | ("(" Ident "subset" "Z" ")") "," (Ident | "{ Ident [ArrayNotation] ["," Ident [ArrayNotation]]) ")" [GroupmemberAssignList].

3.2.3 Definitions

In the definition section there are two types of definitions: One for defining the homomorphisms and another one for defining the integer interval.

Homomorphism Definition The definition of the homomorphism is the key part of the file. The structure of the homomorphism has a lot of implications on other parameters as it will be explained in the following sections. Each homomorphism that was declared has to be defined now in this section.

The definition of a homomorphism starts always with the keyword `DefineHomomorphism`. This keyword is followed by two arguments, the name of the homomorphism that should be defined and the structure of the homomorphism. Please notice that the homomorphism name needs to be declared beforehand in the declaration section.

The compiler provides two levels of abstraction for the definition of the structure of a homomorphism: An abstract way that mentions only the groups that are involved and a concrete way that defines the mapping of an element from the domain into the co-domain.

Abstract Definition of the Homomorphism Structure For the abstract definition of a homomorphism structure, the homomorphism is defined only as a mapping from a tuple of groups called the domain to another tuple of groups known as co-domain. The homomorphic function itself does not have to be specified. This will be called the abstract definition of a homomorphism.

Here is an example of such an abstract definition:

```
DefineHomomorphism(zeta_4, G_1 # Zm # Zm -> H_3 # H_4);
```

The `#` sign stands for the combination of groups and can be used in the domain as well as in the co-domain. Domain and co-domain are separated by the arrow that consists of the two signs `-` and `>`.

If a homomorphism is specified in the abstract way, it will be referred to in the output file always as a blackbox, where the appropriate arguments are given as parameters. Figure 3.9 shows how this looks in the LaTeX output.

$$t_{0,0} := \zeta(r_{0,0}, r_{0,1}, r_{0,2})$$

Figure 3.9: Abstract Homomorphism Structure as written in LaTeX Output

In case of java output, the homomorphism will be referred as an interface with the name of the homomorphism. In order to run the generated proof, the prover and the verifier will need an instantiation of an implementation of this homomorphism as constructor argument.

Concrete Definition of the Homomorphism Structure A homomorphism can be defined not only as an abstract mapping from domain into co-domain but also as a concrete function. Such a concrete definition could look like this.

```
DefineHomomorphism(zeta_2, (x_[2..5]) |-> (h_1^(4*x_2) * h_2^(x_3*5 + x_4), h_3^x_5));
```

The concrete function has to be of the following form: It starts with either a single preimage or with a tuple of preimages that needs enclosing parenthesis and the elements are either comma separated or in array notation. Then the "maps to" sign which is composed of the three characters "`|`", "`-`" and "`>`". On the right hand side of the "maps to" sign there is a tuple of expressions. The number of expressions in this tuple corresponds to the number of groups in the co-domain.

Each expression can use group elements and integer constants that were all declared beforehand and integer numbers[6]. The expression has to be semantically correct. In other words group operations can only be processed to two group elements from the same group, a repeated operation needs as specification of the number of times that the group operation shall be applied either an integer number, an integer constant or a member of an additive group and so on. The operation signs that are used have to correspond with the operation sign defined for the corresponding group in the co-domain that was declared in the declaration section.

If a homomorphism is specified with a concrete structure this implies the following for the output:

- In case of LaTeX code, the concrete function is used throughout the protocol and the preimages from the definition are simply replaced by the actual arguments of this calculation. See figure 3.10 for the example.

$$(t_{0,0}, t_{0,1}) := (h_1^{r_{0,0} \cdot 4} \cdot h_2^{(r_{0,1} \cdot 5 + r_{0,2})}, h_3^{r_{0,3}})$$

Figure 3.10: Concrete Homomorphism Structure as written in LaTeX Output

- In case of java code generation, this homomorphism is created and instantiated as a class of its own that implements the homomorphism interface. Therefore this homomorphism does not have to be implemented by the user himself.

Integer Interval Definition The definition of an integer interval is very simple. All that has to be done is to define the two limits for this interval. The limits have to be integer constants that are declared in the declaration section. Here is an example of the integer interval definition:

```
DefineIntegerInterval(i_1, (a_1,b_1));
```

[6] That do not have to be declared beforehand

EBNF for the Definition Section The EBNF for the definition section is rather complicated because of the definition of arbitrary expressions for the concrete homomorphisms.

Definition ::= HomomorphismDefinition — IntegerIntervalDefinition ";".

HomomorphismDefinition ::= "DefineHomomorphism" HomomorphismDefinitionList.

HomomorphismDefinitionList ::= "(" Ident "," (AbstractStructure | ConcreteStructure) ")" { ", "(" Ident "," (AbstractStructure | ConcreteStructure) ")" }.

ConcreteStructure ::= (Ident | ("(" IdentList ")")) "|-> "(" Expression { "," Expression }.

IdentList ::= Ident [ArrayNotation] ["," IdentList].

Expression ::= TermList.

TermList ::= Term [("+"| "-") TermList].

Term ::= FactorList.

FactorList ::= Factor [("*"| "/") FactorList].

Term ::= ExponentList.

ExponentList ::= Exponent ["^" ExponentList].

Exponent ::= Ident | Number | ("(" Expression ")").

AbstractStructure ::= PreimageGroupStructure "->" ImageGroupStructure.

PreimageGroupStructure ::= (Ident | ("Z" Letter) | InfiniteGroupStructure) [" # " (Ident | ("Z" Letter) | InfiniteGroupStructure)].

InfiniteGroupStructure ::= Ident "subset" "Z".

ImageGroupStructure ::= (Ident | ("Z" Letter)) ["# " (Ident | ("Z" Letter))].

IntegerInterval ::= "DefineIntegerInterval" IntegerIntervalDefinitionList.

IntegerIntervalDefinitionList ::= "(Ident "," "(" Ident "," Ident ")" ")" ["," IntegerIntervalDefinitionList].

3.3 Part 2: The Protocol Specification

In the second part of the input file the protocol instance has to be specified. All these points are part of this specification:

- The mapping of group elements to the specific place in the homomorphism as image or preimage.
- The definition of the image that might be an algebraic expression.
- The specification of an access structure if it is known at compile time. This access structure can be specified either as a set of qualified sets or as a boolean formula.
- In order to choose between the two protocol types Σ protocol or 2Σ protocol the index set has to be defined that contains all those preimages for which an additional homomorphism should be applied.
- The preimages for which an additional interval check should be performed have to be named.
- Constraints between preimages from the same group can be specified.
- Variable names for all internal protocol variables can be set.
- The target language needs to be specified.
- In case of LaTeX code generation two different modes are available.

3.3.1 Relation Specification

The first part in the protocol specification section is called the "Relation" section. In this part the mapping of group elements to homomorphisms is specified and if desired an access structure is given. There are two different syntax variations that differe in the way how the information is specified but in terms of power and expressiveness are equivalent. We will call the first way the enumerative description and the second one the boolean description. The shortcut notation with the array style that has been provided in part 1 is now no more supported.

Enumerative Description In the enumerative description, the relation is broken up into three parts, the common input, the preimage input and optional an access structure. The common input and the preimage input are both an ordered list of tuples that have to correspond in number and position. Each tuple of the common input consists of the name of the homomorphism variable and the list of public known images. Each tuple in the preimage input consists only of the preimages according to the homomorphism that stands in the common input at the same position.

Here is an example of the relation definition in enumerative description:

```
Relation [
CommonInput = {(zeta_1,(y_1)), (zeta_2,(y_2, y_3)),(zeta_3,(y_4)) };
PreimageInput = {(x_1),(x_2, x_3, x_4, x_5),(x_8, x_7)};
AccessStructure = {{zeta_1, zeta_2, zeta_3}};
]
```

The CommonInput and the PreimageInput are specified as a set of elements. The number of elements have to be equal because there is a direct mapping between the tuples according to the order. So the order in which the elements appear does matter. In our example the compiler will interpret the input that way that x_1 is the preimage that belongs to homomorphism ζ_1 with the image y_1.

The Access Structure specification gives a set of sets. Each of these sets is called a qualified set. If a prover knows all preimages according to one of the qualified sets and acts correctly, the verifier will accept the proof provided he acts honestly. The specification of an access structure is optional. If it is not specified, it will just be one of the parameters that have to be given to the protocol at runtime.

Boolean description In boolean description, the relation is given as one single boolean formula. The litterals are the equations where the homomorphism links the preimages with the corresponding images. Here is an example of the relation definition in boolean description:

```
Relation = ([y_1 = zeta_1(x_1)] && [y_2 = zeta_2(x_2)]) || [y_3 = zeta_3(x_3)];
```

This formula may be of any structure using conjunction and disjunction signs as known from programming languages.

For reasons of parsing and efficiency in terms of lookahead symbols, each homomorphism term has to be surrounded by the brackets "[", and "]". The operator preceedence is that "`&&`" has a higher preceedence than "`||`" just as usual.
In the boolean description there is no way to leave the access structure undefined as it is deduced from the boolean formula that is specified.

Arithmetic Expressions on the Images It is possible to specify for the images not only a single group element but a whole algebraic expression. This expression has to be correct in terms of group operation signs from the corresponding groups of the group elements involved in this expression.

```
Relation = [(y_1^(5*d)) = zeta_1(x_1, x_2, x_3)];
```

All the rules that were already mentioned in section 3.2.3 have to be considered as it is exactly the same production in the EBNF grammar that will be applied.

Constraints on the Relation Specification There are several constraints on the relation specification that are checked by the compiler in order to guarantee a correct protocol and that have to be considered by the user. Some of them depend on whether the homomorphisms were specified in an abstract or in a concrete way.

Each homomorphism has a fixed number of preimages and images according to the definition in the definition section. The number of preimages and images must of course correspond with this number. In case of concrete definition of the homomorphism the preimage symbols were already given and so they must correspond now with those specified in the definition section.

Each group element is assigned to a group. In case of abstract homomorphism definition each homomorphism has a well-defined set of groups in the domain and the co-domain. These groups have to match with those that the preimages and images assigned to this homomorphism belong to. If the homomorphism was specified in an concrete way, the groups of the domain and co-domain will be deduced from the preimages and images.

EBNF for the Relation Specification As the syntax for the relation specification is rather complicated mainly because of the boolean description, we will present it separately.

Relation ::= "Relation" EnumerativeDescription | BooleanDescription.

EnumerativeDescription ::= "[" CommonInput PreImageInput [AccessStructure].

CommonInput ::= "CommonInput" "=" "{" CommonInputTuple ["," CommonInputTuple] "}" ";".

CommonInputTuple ::= "(" Ident "," "(" Expression { "," Expression } ")" ")".

PreImageInput ::= "PreimageInput" "=" "{" PreImageInputTuple { "," PreImageInputTuple } "}" ";".

PreImageInputTuple ::= "(" Ident { "," Ident } ")".

AccessStructure ::= "AccessStructure" "=" "{ QualifiedSet { "," QualifiedSet } "}.

QualifiedSet ::= "{" Ident { "," Ident } "}".

BooleanDescription ::= "=" BooleanTermList.

BooleanTermList ::= BooleanTerm [`||` BooleanTermList].

BooleanTerm ::= BooleanFactorList.

BooleanFactorList ::= BooleanFactor [`&&` BooleanFactorList].

BooleanFactor ::= HomomorphismRelation | "(" BooleanTermList ")".

HomomorphismRelation ::= "[" ImageHomRelationTuple "=" Ident "(" PreImageHomRelationTuple ")" "].

PreImageHomRelationTuple ::= Ident { "," Ident }

ImageHomRelationTuple ::= "(" Expression ")".

Expression ::= TermList.

TermList ::= Term [("+"| "-") TermList].

Term ::= FactorList.

FactorList ::= Factor [("*"| "/") FactorList].

Term ::= ExponentList.

ExponentList ::= Exponent ["^" ExponentList].

Exponent ::= Ident | Number | ("(" Expression ")").

3.3.2 Constraints on preimages

On preimages from the same group there might be constraints e.g. `x_2 = 3*x_3`. That means that for the preimages and for all temporary variables based on these preimages that are used during the protocol this relation has to hold.

The constraint has on the left hand side a single preimage without any coefficients. On the right hand side there might be several preimages each with a coefficient combined with the operation of the group these preimages belong to. This group operation sign was specified in the declaration section. The coefficients have to be integer numbers.

Here is an example for such a constraint declaration:

```
Constraints = (x_2 = 3*x_3) && (x_4 =  -3*x_2 + x_3) && (x_8 = x_9);
```

All constraints have to be connected with the `&&` sign. It is not possible to specify an arbitrary boolean formula. There are several conditions that each constraint declaration has to fulfill:

- All preimages in one constraint have to belong to the same group. For the additive infinite group there is an additional requirement: All preimages have be declared from the same integer interval.
- The operation sign has to be corresponding to the group operation sign. If the group operation sign is "+", a constraint could look like this: `x_2 = 3*x_3 + 2*x_4`. If the group operation sign is "*" the same constraint would look like this: `x_2 = x_3^3 * x_4^2`.
- The order in which the constraints are declared is important in order to avoid any cyclic dependencies. A preimage on the left hand side of a constraint may not appear on the right hand side of a constraint that is earlier in the left-to-right order. The assignment of dependent variables will therefore be processed from left to right.

  Here is an example of an illegal order of constraint declarations:

  ```
  Constraints = (x_1 = 2* x_3) && (x_3 = 6*x_5);
  ```

- All constraints have to be compliant with the access structure if one was defined. If that is the case, the following has to hold. First we compute the set of involved homomorphisms for each constraint. Then we compare this set of homomorphism with each qualified set in the access structure. Now the constraint declaration is legal if for each of these comparisons it holds that either the set from the constraint is a subset of the qualified set or the intersection between both sets is empty.

The whole point with the constraint declaration is optional and can be left out.

Implicit Constraints Additional to the constraints that are specified explicitely in the constraint section, there might be implicit constraints. This is the case if the same preimage is used in several homomorphisms. Here is an example of such a specification:

```
Relation = [(y_1) = zeta_1(x_1, x_2, x_3)] && [(y_2) = zeta_2(x_4, x_1)];
```

The preimage `x_1` is used in both homomorphism. Therefore the compiler creates a constraint on the two occurences of `x_1`. The effect is exactly the same as if it would have been written like that:

```
Relation = [(y_1) = zeta_1(x_1, x_2, x_3)] && [(y_2) = zeta_2(x_4, x_5)];
Constraints = (x_1 = x_5);
```

The implicit constraints have to follow exactly the same rules in connection with the access structure as the implicit ones.

EBNF for the Constraints on Preimages

Constraints ::= "Constraints" "=" ConstraintEquation { `&&` ConstraintEquation } ";".

ConstraintEquation ::= "(" Ident "=" ConstraintTuple { ("+"| "-" | "*" | "/") ConstraintTuple } ")".

ConstraintTuple ::= ["-"] (Number "*" Ident) — (Ident "^" Number).

3.3.3 Index Set

There are protocol instances for which the Σ protocol is no proof-of-knowledge depending on the groups that are involved. For these cases the 2Σ protocol can be used. All those preimages for which the 2Σ protocol with its additional homomorphism should be applied have to be included in an index set.

The syntax for the index set definition is very simple:

```
IndexSet = {x_1, x_2, x_7};
```

For the definition of the index set, the order of the preimages is of no importance. The only constraint on the preimage is that their corresponding group is either the additive infinte group $\mathbb{Z}$ or an additive finite group $\mathbb{Z}_m$ that was declared as `Zm` with an arbitrary letter instead of the "m" that stands for the order of the group.

The index set definition can be left out completely if no preimage should be included.

EBNF for the Index Set Definition

IndexSet ::= "IndexSet" "=" "{" Ident ["," Ident] "} ";".

3.3.4 Interval Checks

For all preimages from the additive infinite group it is an additional option to check whether the values computed during the protocol still are in the correct interval.
The syntax for the interval check definition is as follows:

```
IntervalCheck = {x_1, x_6};
```

The only constraint on including preimages into the set for the interval checks is that the preimage belongs to the additive infinite group $\mathbb{Z}$ and has therefore an integer interval assigned with it.

The interval check options is facultative.

EBNF for the Interval Check Definition

IntervalCheck ::= "IntervalCheck" "=" "{" Ident ["," Ident] "} ";".

3.3.5 Changing default Variable Names

Each input element such as homomorphisms, groups, group elements, interval limits and so on can be given a name that is chosen by the user. This name will be used througout the protocol, in the latex output as well as in the java output. But not only those elements that have to be given to the protocol from an external source but also all temporary variables and instances of primitives such as the secret sharing scheme used and created by the compiler itself can be given names by the user.

Here is an example for this optional section in the input file:

```
ParameterNames {
SRSAModulus = n;
SRSAGenerator = g;
SRSAExponent = rho;
VariableR = r;
VariableS = s;
VariableT = t;
VariableC = c;
ChallengeMax = c_plus;
SecurityParameterSZK = sigma_z;
SecurityParameterSZK = sigma_z;
SecurityParameterSZK = sigma_z;
AccessStructure = Gamma;
SecretSharingScheme = Theta;
QualifiedSet = A;
VariableDeltaS = sigma;
VariableDeltaC = gamma;
}
```

All these elements have default names that can be found in the table in section 3.3.5. The default names will be used if they were not explicitely overwritten.

There are some conventions about those names. For example will a greek letter name like `zeta` in the latex code be shown as the greek literal, in this case as ζ. Names with indices such as `s_i` will be written in latex with subscript as s_i. The explicit listing of those features can be found in the usermanual too.

Default Names of different Protocol Elements All the elements used for the protocol which are not directly influenced through the common or the private input have default variable names. These names can be changed by the user by assigning new variable names in the "DefineProtocol" section of the input file.
The default names are set as follows:

Protocol Element	Default name
AccessStructure	`Gamma`
ChallengeMax	`c_plus`
QualifiedSet	`A`
SecretSharingScheme	`Theta`
SecurityParameterSRSA	`sigma_v`
SecurityParameterSZK	`sigma_z`
SecurityParameterValidity	`sigma_s`
SRSAExponent	`rho`
SRSAGenerator	`g`
SRSAModulus	`m`
VariableC	`c`
VariableR	`r`
VariableS	`s`
VariableT	`t`
VariableX	`x`
VariableY	`y`
VariableDeltaS	`sigma`
VariableDeltaC	`gamma`

EBNF for Changing the Variable Names

VariableNames ::= "ParameterNames" "{" VariableNameDefinition "}".

VariableNameDefinition ::= "AccessStructure" "=" Ident ";" | "ChallengeMax" "=" Ident ";" | "QualifiedSet" "=" Ident ";" | "SecretSharingScheme" "=" Ident ";" | "SecurityParameterSRSA" "=" Ident ";" | "SecurityParameterSZK" "=" Ident ";" | "SecurityParameterValidity" "=" Ident ";" | "SRSAExponent" "=" Ident ";" | "SRSAGenerator" "=" Ident ";" | "SRSAModulus" "=" Ident ";" | "VariableC" "=" Ident ";" | "VariableR" "=" Ident ";" | "VariableS" "=" Ident ";" | "VariableT" "=" Ident ";" | "VariableX" "=" Ident ";" | "VariableY" "=" Ident ";". | "VariableDeltaS" "=" Ident ";". | "VariableDeltaC" "=" Ident ";". |

3.3.6 Target Specification and Layout Mode

The last mandatory point in the file is the specification of the target language that should be generated — either java or latex.

In case of the LaTeX code generation there are two possibilities that vary in the level of detailing. On one side there is the layout mode "compact" that shows the overall structure but hides the details about the execution of the protocol. On the other side is the mode "verbose" that shows every single operation that has to be performed.

The syntax for the targed decision is like this:

```
Target = JAVA;
```

or

```
Target = LATEX;
```

In case of LaTeX code generation the optional point that might follow is

```
Layout = COMPACT;
```

If the compact mode was not explicitly specified, the protocol instance will be generated in verbose mode.

EBNF for the Target Choice

Target ::= "Target" "=" "JAVA" | "LATEX" ";".

Layout ::= "Layout" "=" "COMPACT" ";"

3.4 EBNF Syntax Definition of the Input Language

This is the complete EBNF Syntax Definition of the compiler's input language:

Input ::= Declaration { Declaration } Definition { Definition } Assignment { Assignment } Protocol.

Declaration ::= GroupDeclaration | ((“GroupElement” | “Homomorphism” | “IntegerInterval” | “IntegerConstant”) IdentList)“;”.

GroupDeclaration ::= “Group” “(” Ident [ArrayNotation] “,” (“+” | “*”) “)” [“,” “(” Ident [ArrayNotation] “,” (“+” | “*”) “)”].

IdentList ::= Ident [ArrayNotation] { “,” Ident [ArrayNotation]}

ArrayNotation ::= “[” Number “.” “.” Number “]”.

Ident ::= Letter { Letter | Number | “_” }.

Letter ::= “A” | “B” | ... | “Z” | “a” | ... | “z”.

Number ::= “0” | ... | “9” { “0” | ... | “9” }.

Definition ::= HomomorphismDefinition — IntegerIntervalDefinition “;”.

HomomorphismDefinition ::= “DefineHomomorphism” HomomorphismDefinitionList.

HomomorphismDefinitionList ::= “(” Ident “,” (AbstractStructure | ConcreteStructure) “)” { “, ”(” Ident “,” (AbstractStructure | ConcreteStructure) “)” }.

ConcreteStructure ::= (Ident | (“(” IdentList “)“)) “|-> “(” Expression { “,” Expression }.

IdentList ::= Ident [ArrayNotation] [“,” IdentList].

Expression ::= TermList.

TermList ::= Term [(“+”| “-”) TermList].

Term ::= FactorList.

FactorList ::= Factor [(“*”| “/”) FactorList].

Term ::= ExponentList.

ExponentList ::= Exponent [“^” ExponentList].

Exponent ::= Ident | Number | (“(” Expression “)”).

AbstractStructure ::= PreimageGroupStructure “->” ImageGroupStructure.

PreimageGroupStructure ::= (Ident | (“Z” Letter) | InfiniteGroupStructure) [“ # ” (Ident | (“Z” Letter) | InfiniteGroupStructure)].

InfiniteGroupStructure ::= Ident “subset” “Z”.

ImageGroupStructure ::= (Ident | (“Z” Letter)) [“# ” (Ident | (“Z” Letter))].

IntegerInterval ::= “DefineIntegerInterval” IntegerIntervalDefinitionList.

IntegerIntervalDefinitionList ::= “(Ident “,” “(” Ident “,” Ident “)” “)” [“,” IntegerIntervalDefinitionList].

Assignment ::= “AssignGroupMember” GroupMemberAssignmentList “;”.

GroupMemberAssignmentList ::= “(” Ident | (“(” Ident “subset” “Z” “)”) “,” (Ident | “{ Ident [ArrayNotation] [“,” Ident [ArrayNotation]]) “)”[GroupmemberAssignList].

Protocol ::= “SpecifyProtocol” “[” Relation [Constraints] [IndexSet] [IntervalCheck] [VariableNames] Target [Layout] “]”.

Relation ::= “Relation” EnumerativeDescription | BooleanDescription.

EnumerativeDescription ::= “[” CommonInput PreImageInput [AccessStructure].

CommonInput ::= “CommonInput” “=” “{” CommonInputTuple [“,” CommonInputTuple] “}” “;”.

CommonInputTuple ::= “(” Ident “,” “(” Expression { “,” Expression } “)” “)”.

PreImageInput ::= “PreimageInput” “=” “{” PreImageInputTuple { “,” PreImageInputTuple } “}” “;”.

PreImageInputTuple ::= “(” Ident { “,” Ident } “)”.

AccessStructure ::= “AccessStructure” “=” “{ QualifiedSet { “,” QualifiedSet } “}.

QualifiedSet ::= “{” Ident { “,” Ident } “}”.

BooleanDescription ::= “=” BooleanTermList.

BooleanTermList ::= BooleanTerm [|| BooleanTermList].

BooleanTerm ::= BooleanFactorList.

BooleanFactorList ::= BooleanFactor [&& BooleanFactorList].

BooleanFactor ::= HomomorphismRelation | “(” BooleanTermList “)”.

HomomorphismRelation ::= “[” ImageHomRelationTuple “=” Ident “(” PreImageHomRelationTuple “)” “].

PreImageHomRelationTuple ::= Ident { “,” Ident }

ImageHomRelationTuple ::= “(” Expression “)”.

Constraints ::= “Constraints” “=” ConstraintEquation { && ConstraintEquation } “;”.

ConstraintEquation ::= “(” Ident “=” ConstraintTuple { (“+”| “-” | “*” | “/”) ConstraintTuple } “)”.

ConstraintTuple ::= ["-"] (Number "*" Ident) — (Ident "^" Number).

IndexSet ::= "IndexSet" "=" "{" Ident ["," Ident] "} ";".

IntervalCheck ::= "IntervalCheck" "=" "{" Ident ["," Ident] "} ";".

VariableNames ::= "ParameterNames" "{" VariableNameDefinition "}".

VariableNameDefinition ::= "AccessStructure" "=" Ident ";" | "ChallengeMax" "=" Ident ";" | "QualifiedSet" "=" Ident ";" | "SecretSharingScheme" "=" Ident ";" | "SecurityParameterSRSA" "=" Ident ";" | "SecurityParameterSZK" "=" Ident ";" | "SecurityParameterValidity" "=" Ident ";" | "SRSAExponent" "=" Ident ";" | "SRSAGenerator" "=" Ident ";" | "SRSAModulus" "=" Ident ";" | "VariableC" "=" Ident ";" | "VariableR" "=" Ident ";" | "VariableS" "=" Ident ";" | "VariableT" "=" Ident ";" | "VariableX" "=" Ident ";" | "VariableY" "=" Ident ";".

Target ::= "Target" "=" "JAVA" | "LATEX" ";".

Layout ::= "Layout" "=" "COMPACT" ";"

4 Generated Output Files

The backend of the compiler actually consists of two different code generators, one that generates a specification written in latex code and one that generates executable java source code. In this section I want to give a short overview of the two different artefacts that can be generated.

4.1 Latex Files

The LaTeX file consists of two parts. The first one mirrors the declarations and definitions that were given in the input file and shows the protocol parameters that were chosen. The second part is the protocol execution which is split up in different rounds.

4.1.1 Declaration and Definition Section

This section is similar to the input file. It starts with all homomorphisms used for this protocol. If one of the groups in the domain is infinite, two more functions are added which illustrate the mapping for the secret x into the co-domain and the mapping for the protocol variables r and s. These two functions which are called the same as the homomorphism function e.g. ζ but a prime resp. two primes are added e.g. ζ' resp. ζ''. Figure 4.11 shows these additional function declarations.

Homomorphism ζ

$$\zeta : G_1 \times \mathbb{Z}^2 \times G_2 \to H,$$

$$G'_\zeta = G_1 \times [a,b]^2 \times G_2$$
$$G''_\zeta = G_1 \times [-2^{\sigma_z} c^+ \lceil \tfrac{b-a}{2} \rceil, 2^{\sigma_z} c^+ \lceil \tfrac{b-a}{2} \rceil]^2 \times G_2$$
$$\zeta' = \zeta | G'_\zeta$$
$$\zeta'' = \zeta | G''_\zeta$$

Figure 4.11: Declaration Section of Homomorphisms with Infinite Groups

If preimages from the homomorphism are included in the index set that specifies the set of variables where the 2Σ protocol is applied, an additional entry called index set is associated to the homomorphism. This entry lists the preimages for which an additional homomorphism is created.

If some of the user defined homomorphisms have to be mapped together into atoms, the list of the atoms during the protocol execution will be displayed too.. The reason for performing such a mapping are described in section 5.7.2 in detail.Figure 4.12 shows this part of the declaration section.

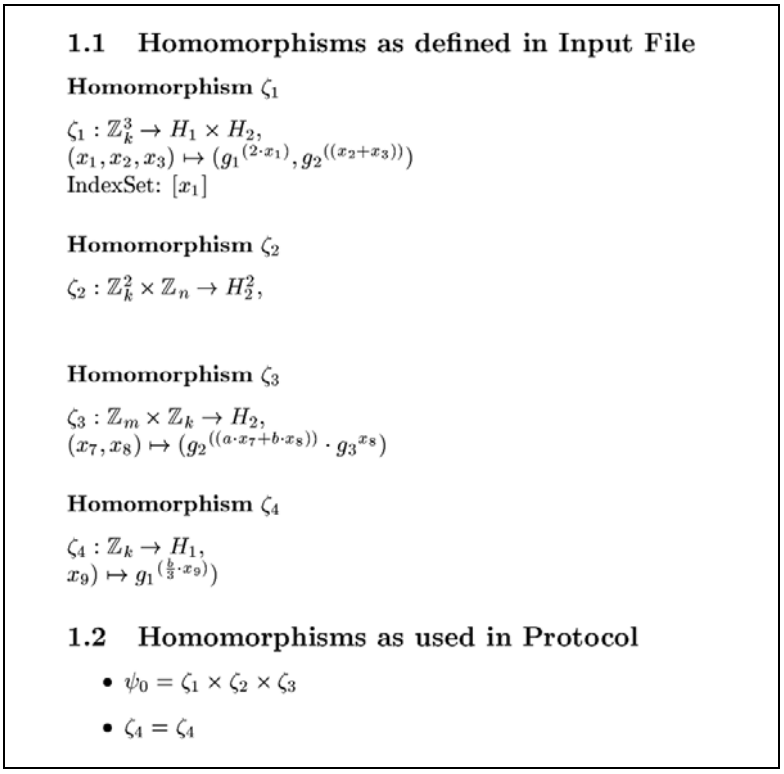


Figure 4.12: User defined Homomorphisms and the Mapping into Atoms as shown in the LaTeX output

The next item is the access structure which is displayed in DNF, where each clause represents one of the qualified sets. In case where no access structure is specified [7], there is simply no information that could be displayed. In that case, the access structure is one of the common input elements and will be listed there. Figure 4.13 shows this part of the declaration section.

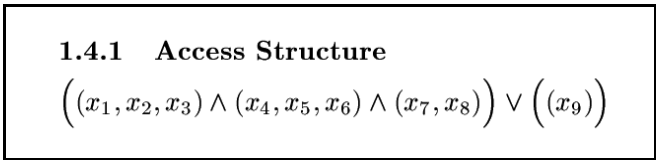


Figure 4.13: The Access Structure Section in the Declaration part of the LaTeX output

Another optional point is the list of preimages for which interval checks should be performed. Figure 4.14 shows the interval check section of the declaration.

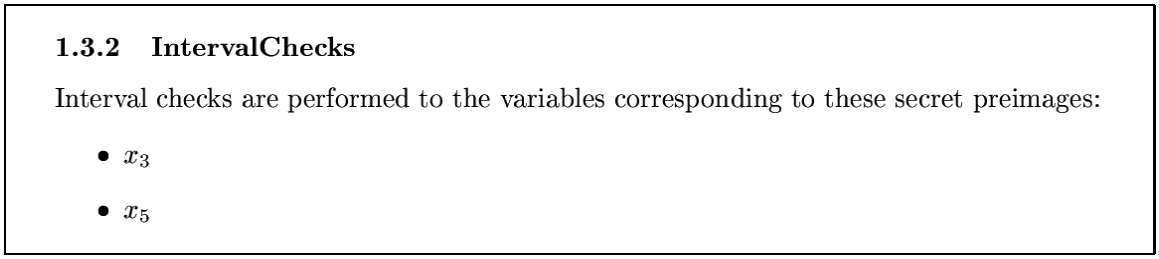


Figure 4.14: The Interval Checks Section in the Declaration part of the LaTeX output

If constraints on preimages are either defined explicitely or implicitely as explained in section 3.3.2 by using the same preimage in different homomorphisms, all these constraints will be listed.Figure 4.15 shows the interval check section of the declaration.

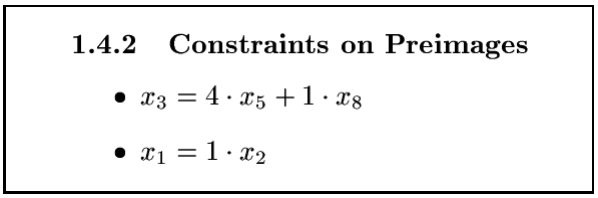


Figure 4.15: The Constraints Section in the Declaration part of the LaTeX output

The relation between the images and the preimages is shown in the relation section. This section consists of one equation per homomorphism.

[7] That could be the case if the relation was specified in the input file using the enumerative description as described in section 3.3.1

1.5 Relation

- $\zeta_1 : (y_1{}^d, y_2) = (g_1{}^{(2 \cdot x_1)}, g_2{}^{((x_2 + x_3))})$
- $(y_2, y_3) = \zeta_2(x_4, x_5, x_6)$
- $\zeta_3 : y_2{}^4 = g_2{}^{((a \cdot x_7 + b \cdot x_8))} \cdot g_3{}^{x_8}$
- $\zeta_4 : y_1 = g_1{}^{(\frac{b}{3} \cdot x_9)}$

Figure 4.16: The Relation Section in the Declaration part of the LaTeX output

As a kind of conclusion from the different properties that can be derived from the whole protocol distribution in case that the verifier accepts the proof, an additional section will be displayed. This section shows up only if an access structure is specified. It takes into account the collision equation from the different homomorphisms, the interval checks and the constraints. Figure 4.17 shows this kind of conclusion.

2 Protocol Properties

$$\Big[\Big(\big(\tfrac{{P_{UO}}^2}{{wT_O}^{(2 \cdot a_3 \cdot r_{5UO})} \cdot {z_O}^{(2 \cdot a_4 \cdot r_{6UO})}}\big)^{\gamma_0} =$$
$${a_O}^{(2 \cdot \alpha)} \cdot {w_O}^{(2 \cdot \beta_5)} \cdot \big({v_{O,1}}^{(2 \cdot a_1)}\big)^{\beta_1} \cdot \big({v_{O,2}}^{(2 \cdot a_1)}\big)^{\beta_2} \cdot \big({v_{O,3}}^{(2 \cdot a_1)}\big)^{\beta_3} \cdot \big({vT_O}^{(2 \cdot a_2)}\big)^{\beta_4} \cdot \big({b_O}^2\big)^{\gamma}\Big)$$
$$\wedge \Big((C^2 \cdot {b_O}^{(2 \cdot sT)})^{\gamma_0} = \big({v_{O,1}}^{(2 \cdot a_1)}\big)^{\beta_{P,1}} \cdot \big({v_{O,2}}^{(2 \cdot a_1)}\big)^{\beta_{P,2}} \cdot \big({v_{O,3}}^{(2 \cdot a_1)}\big)^{\beta_{P,3}} \cdot \big({vT_O}^{(2 \cdot a_2)}\big)^{\beta_{P,4}} \cdot \big({b_O}^2\big)^{\gamma} \cdot \big({h_O}^2\big)^{\epsilon}\Big)$$
$$\wedge \Big((g^{rT_1})^{\gamma_0} = (g^{-1})^{\beta_{Pt,1}} \cdot g^{\beta_{t,1}}\Big) \wedge \Big((g^{rT_2})^{\gamma_0} = (g^{-1})^{\beta_{Pt,2}} \cdot g^{\beta_{t,2}}\Big) \wedge \Big((g^{rT_3})^{\gamma_0} = (g^{-1})^{\beta_{Pt,3}} \cdot g^{\beta_{t,3}}\Big)$$
$$\wedge \Big((g^{rT_4})^{\gamma_0} = (g^{-1})^{\beta_{Pt,4}} \cdot g^{\beta_{t,4}}\Big) \wedge \Big((K_{UO})^{\gamma_0} = g^{\alpha_t} \cdot h^{\beta_{t,5}}\Big) \wedge \Big(\beta_{P,1} = 1 \cdot \beta_{Pt,1}\Big) \wedge \Big(\beta_{P,2} = 1 \cdot \beta_{Pt,2}\Big)$$
$$\wedge \Big(\beta_{P,3} = 1 \cdot \beta_{Pt,3}\Big) \wedge \Big(\beta_{P,4} = 1 \cdot \beta_{Pt,4}\Big) \wedge \Big(\beta_1 = 1 \cdot \beta_{t,1}\Big) \wedge \Big(\beta_2 = 1 \cdot \beta_{t,2}\Big) \wedge \Big(\beta_3 = 1 \cdot \beta_{t,3}\Big) \wedge \Big(\beta_4 = 1 \cdot \beta_{t,4}\Big)$$
$$\wedge \Big(\alpha_t = 1 \cdot \alpha\Big) \wedge \Big(\gamma_{\zeta_2} = 1 \cdot \gamma_{\zeta_1}\Big) \wedge \Big(\sigma_{0,0} \in [-2^{(\sigma_z+2)}c^{+\frac{b_2-a_2}{2}}, 2^{(\sigma_z+2)}c^{+\frac{b_2-a_2}{2}}]\Big)$$
$$\wedge \Big(\sigma_{0,1} \in [-2^{(\sigma_z+2)}c^{+\frac{b_2-a_2}{2}}, 2^{(\sigma_z+2)}c^{+\frac{b_2-a_2}{2}}]\Big) \wedge \Big(\sigma_{0,2} \in [-2^{(\sigma_z+2)}c^{+\frac{b_2-a_2}{2}}, 2^{(\sigma_z+2)}c^{+\frac{b_2-a_2}{2}}]\Big)$$
$$\wedge \Big(\sigma_{0,3} \in [-2^{(\sigma_z+2)}c^{+\frac{b_2-a_2}{2}}, 2^{(\sigma_z+2)}c^{+\frac{b_2-a_2}{2}}]\Big) \wedge \Big(\sigma_{0,12} \in [-2^{(\sigma_z+2)}c^{+\frac{b_1-a_1}{2}}, 2^{(\sigma_z+2)}c^{+\frac{b_1-a_1}{2}}]\Big)$$
$$\wedge \Big(\sigma_{0,6} \in [-2^{(\sigma_z+2)}c^{+\frac{b_1-a_1}{2}}, 2^{(\sigma_z+2)}c^{+\frac{b_1-a_1}{2}}]\Big) \wedge \Big(\sigma_{0,11} \in [-2^{(\sigma_z+2)}c^{+\frac{b_1-a_1}{2}}, 2^{(\sigma_z+2)}c^{+\frac{b_1-a_1}{2}}]\Big) \wedge \Big(\gamma_0 | \sigma_{0,0}\Big)$$
$$\wedge \Big(\gamma_0 | \sigma_{0,4}\Big) \wedge \Big(\gamma_0 | \sigma_{0,5}\Big)\Big]$$

Figure 4.17: The Constraints Section in the Declaration part of the LaTeX output

The declaration and definition section ends with two mandatory points, the common input with all those elements that have to be known to the both entities, the prover and the verifier, and the preimage input that should be known to the prover only. As it is impossible to know at compile

time which of the qualified sets from the access structure will be known to the prover, all preimages are listed although it is not needed that the prover knows all of them. Figure 4.18 shows this kind of conclusion.

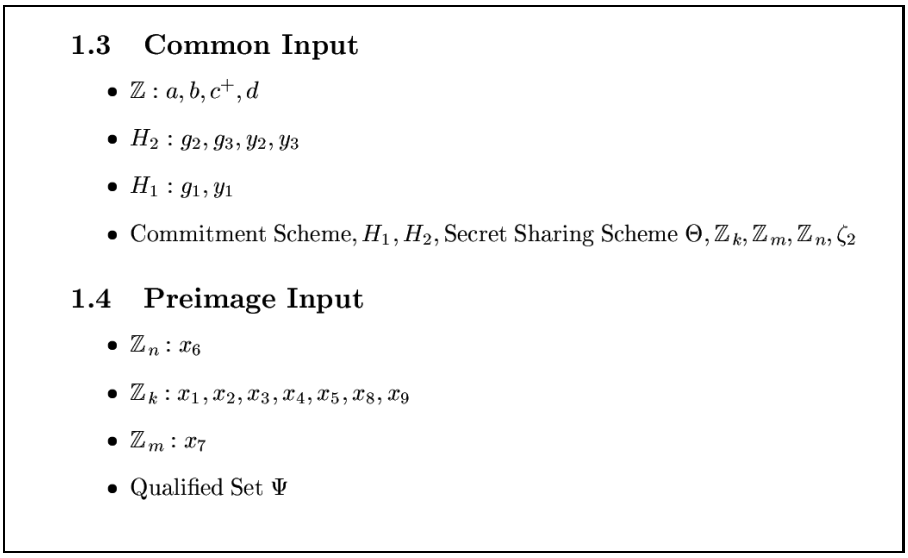


Figure 4.18: The Common Input and the Preimage Input Section from the Declaration Part of the LaTeX output

4.1.2 Protocol

The protocol section consists of the different rounds of the protocol execution. Their number depends on the protocol type.

On the left hand side of each page are those actions that are performed by the prover, the ones of the verifier are on the right. Each send action is shown by an arrow with the data that is being sent written above the arrow.

The protocol works in a synchronous model which is a prerequisite for using the term of rounds, as each round including the send action if it contains one, has to be completed as precondition for the follwing round.

There are two different modes for the user to choose how the protocol should be displayed. The verbose mode lists each single operation explicitely for all homomorphism and each single variable that takes part. As a protocol of some complexity will use a complex homomorphism structure, this protocol specification could get a size that is no more useful for understanding the main structure. Therefore a second mode called the compact mode is provided.

Verbose Mode In the verbose mode each operation is listed explicitely. A protocol written in verbose mode provides the possibiliy of checking each single operation. It uses and shows all given

information about the structure of homomorphisms and groups e.g. the number of groups in the domain and co-domain or the special case where infinite groups are involved.

A protocol written in verbose mode in LaTeX is the direct mapping from the java file that would be generated with the same input in a more readable fashion. All variable names and operations are the same and even in the same order.

Figure 4.19 shows a part of a protocol as displayed in verbose mode.

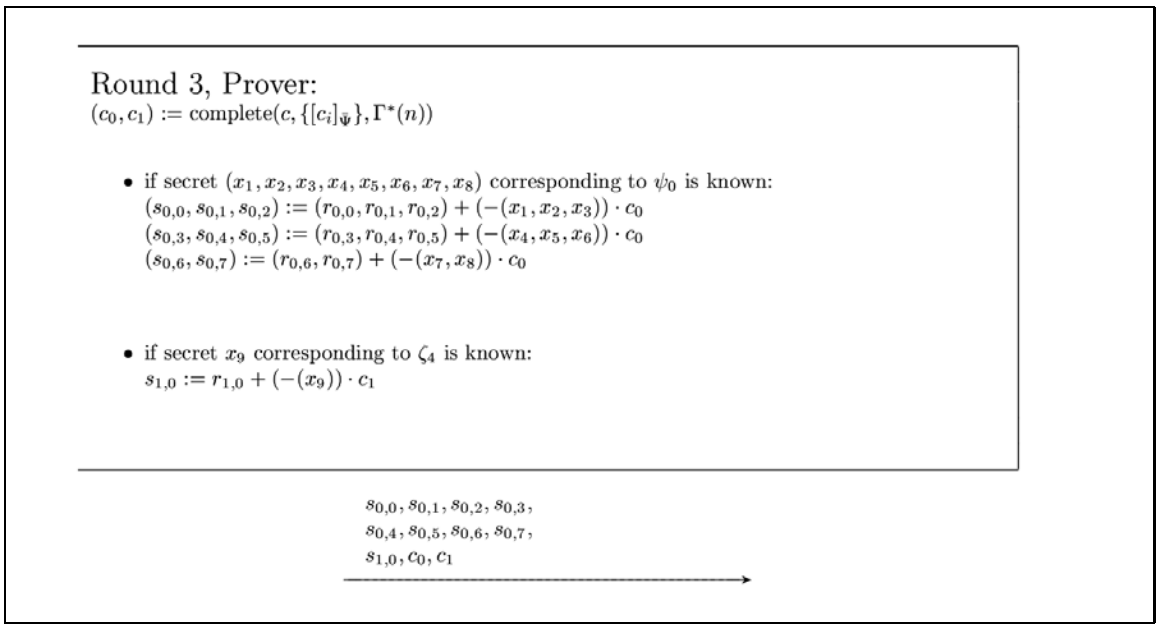


Figure 4.19: Part of a Protocol as displayed in verbose Mode.

Compact Mode The compact mode hides all details about the number and the structure of the homomorphisms used. Each operation is written as an iteration over all homomorphism and only in an abstract way with variables using the actual homomorphism as index.

The compact mode is useful for showing the overall structure of a protocol without being interested in every single operation that has to be performed.

Figure 4.20 shows a part of a protocol as displayed in compact mode.

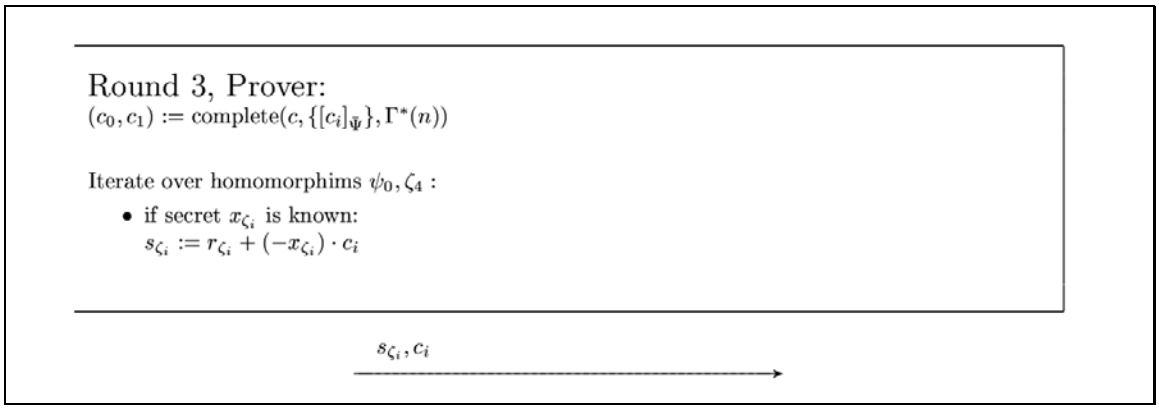


Figure 4.20: Part of a Protocol as displayed in compact Mode.

4.1.3 Additional Packages used for Generating PostScript Files

In order to make the output as readable and the comiler as powerful and flexible as possible I have used some constructs that might not be included in the standart LaTeX distribution. A first package is the package `chemarrow` [8], together with the fonts `arrow.mf` and `arrow.tmf` from the same origin. Furthermore I use the package `longtable` because the length of a round depends on the number and the structure of the homomorphisms and can therefore get a length that does not fit on a single page. So it needs to be stretched over several pages. This package is usually included in the standard distribution. Otherwise it can be found at http://www.ctan.org.

[8] This package can be found at http://ftp.agh.edu.pl/pub/tex/macros/latex/contrib/chemarrow/

4.2 Java Files

The generated java source code is split up into several pieces. The two main elements are the prover and the verifier which are each in a class of its own. All input - either the private and the common input in case of the prover or only the common input in case of the verifier - are given as constructor arguments.

The source code works on interfaces and abstract classes that are defined in the packages called "`interfaces`" and "`abstractions`".[9] In order to instantiate these classes the user has to provide implementations of these interfaces. This mechanism guarantees a very high flexibility in using different kinds of implementations that might be written somewhen later in time that can be used even without recompiling the input file.

In case of the prover class, the constructor asks for all preimages even though an access structure is defined, because at compile time it is unknown which of the qualified sets the prover will know. The constructor itself will just figure out from the preimages that he receives, if it is really one of the qualified sets and if yes, which one. All preimages that are unknown to the prover can be passed just as `null` values.

The protocol will work like this: Both, the prover and the verifier provide a number of methods that correspond to the different rounds. They take as input the parameters that were sent at the end of the last round and return the parameters that have to be sent for the next round. In order to run the protocol, one just has to call one method after the other, always switching from the prover to the verifier.

Figure 4.21 shows the constructor signature of the prover, figure 4.22 the constructor signature of the verifier and figure 4.23 how these could be instantiated and run.

[9] There are some cases where concrete implementations are used in the generated files. This is for example the case for the additional homomorphism that is used for the 2Σ protocol and that is already known at compile time.

```
public class Prover {

    public Prover(
        FiniteGroup G_1,
        FiniteGroup H_1,
        SecretSharingScheme Theta,
        AdditiveGroupElement cPlus,
        GroupElement y_1,
        GroupElement y_2,
        AbstractHomomorphism zeta_1,
        AbstractHomomorphism zeta_2,
        GroupElement x_1,
        GroupElement x_2,
        GroupElement x_3) {
        ...
```

Figure 4.21: Signature of the Prover Class Constructor

```
public class Verifier{

    public Verifier(
        FiniteGroup G_1,
        FiniteGroup H_1,
        SecretSharingScheme Theta,
        AdditiveGroupElement cPlus,
        GroupElement y_1,
        GroupElement y_2,
        AbstractHomomorphism zeta_1,
        AbstractHomomorphism zeta_2) {
        ...
```

Figure 4.22: Signature of the Verifier Class Constructor

```
SecretSharingScheme theta;
...
Prover P = new Prover(theta, c_max, y_1, y_2, hom_1, hom_2, null, x_2, null);
Verifier V = new Verifier(theta, c_max, y_1, y_2, hom_1, hom_2);
try {
    Round1Parameter r1 = P.executeRound1();
    Round2Parameter r2 = V.executeRound2(r1);
    Round3Parameter r3 = P.executeRound3(r2);
    V.executeRound4(r3);
} catch (Exception e) {
    e.printStackTrace();
}
```

Figure 4.23: Instantiation and Execution of the Protocol

4.2.1 Parameter Classes

For each send action a parameter class is generated by the compiler. Another way for solving this problem would have been simply to use a standard collection where all items, that have to be sent, are stored into. The important advantage of my solution is that it guarantees a kind of type safety at compile time. No prover and verifier that have been created using different input parameters may be connected together in a class in order to execute the protocol because the parameters that have to be send would not correspond. This would turn out already at compile time because of the parameter class that would be used to pass these parameters. Figure 4.24 shows an example of an automatically generated parameter class.

```
public class Round1Parameter {
    private GroupElement t_0_0;
    private GroupElement t_1_0;
    private GroupElement t_2_0;

    public Round1Parameter (
     GroupElement t_0_0,
     GroupElement t_1_0,
     GroupElement t_2_0    ) {
        this.t_0_0 = t_0_0;
        this.t_1_0 = t_1_0;
        this.t_2_0 = t_2_0;
    }

    public GroupElement getT_0_0() {
        return t_0_0;
    }
    public GroupElement getT_1_0() {
        return t_1_0;
    }
    public GroupElement getT_2_0() {
        return t_2_0;
    }
}
```

Figure 4.24: Example of an automatically generated Class for Parameter Passing

4.2.2 Concrete Homomorphisms

If a homomorphism is defined as a concrete function, the compiler will generate an implementation of this homomorphism in a class of its own. The class extends the abstract class `AbstractHomomorphism` that implements the `Homomorphism` interface.

The prover and the verifier class will instantiate these concrete homomorphism classes in the constructor. Therefore it will no longer be one of the constructor arguments as it would have been in case of abstract homomorphism definition.

Figure 4.25 shows an example of a concrete homomorphism class.

```
public class Homomorphism_zeta_3 extends AbstractHomomorphism {

    private AdditiveInfiniteGroup Z = new AdditiveInfiniteGroupImpl();
    private FiniteGroup H_1;
    private AdditiveFiniteGroup Zk;
    private AdditiveFiniteGroup Zm;
    private GroupElement g_1;

    public Homomorphism_zeta_3(
        FiniteGroup H_1,
        AdditiveFiniteGroup Zk,
        AdditiveFiniteGroup Zm,
        GroupElement g_1) {

// assignment of groups
    }

    public Iterator image(Iterator it){
        Collection result = new ArrayList();
        AdditiveGroupElement x_4 = (AdditiveGroupElement )it.next();
        AdditiveGroupElement x_7 = (AdditiveGroupElement )it.next();
        GroupElement expr_var1 = H_1.repeatedOperation(g_1, x_4.getValue());
        GroupElement expr_var2 = H_1.repeatedOperation(g_1, x_7.getValue());
        GroupElement expr_var0 = H_1.operate(expr_var1, expr_var2);
        result.add(expr_var0);
        return result.iterator();
    }
}
```

Figure 4.25: Example of an automatically generated Concrete Homomorphism Implementation

5 Design and Implementation

In this section I will give an overview of the design of the compiler. I will explain how it is based on the foundations of compiler construction and describe the different points where it deviates from the common compiler design or rather where it extends this design. I will explain the reasons for the design decisions made in this project and their impact on the zero-knowledge compiler.

Some implementation details that are of special interest will be mentioned, either for reasons of understanding parts of the source code that are not that trivial or simply because the solution involved some additional concepts. These implementation details are explained in the subsections of the component where they belong except for some interesting problems that do not belong to one single section in the compiler design and are therefore mentioned in a subsection of their own in section 5.7.

5.1 Main Components

A compiler as a translator takes input in a certain form and translates it into some different representation.

The basic approach as presented for example by Niklaus Wirth in [8] is to translate each structure directly from the source language into the target language, if possible in a single pass. The main emphasis in this approach is very much on the efficiency.

A second approach in writing compilers is described for example by Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman in [1]. This approach does not try to find a direct mapping from the source language to the target language but the main emphasis here lies on the uncoupling of the two languages. This approach is useful especially for compilers that do not translate only from one language into another but that provide translations from several source into several target languages with the same content.

We have chosen the second approach as the compile time itself for this project is not one of the critical resources. As the compiler should be able to produce two different types of target code, the mechanism to uncouple the soure and the target language is exactly what we need.

For this approach with the aim to uncouple the two representations — the one from the source and the one from the target language — all information that is needed has to be distilled and stored in a form that is independent of the languages or representations it comes from or should be translated into. This form is usually called the intermediate representation and could be any kind of structure which is able to store the information for this specific content.

The first step of a compiler is to extract this information from the input file, building up the intermediate representation. The input file might be written in any well-defined source language. This part of the compiler is called the front end.

In order to get the translation of the information stored in the input file, a second step has to be performed which takes the intermediate representation and maps it into the chosen target language.

This part of the compiler is called the back end.

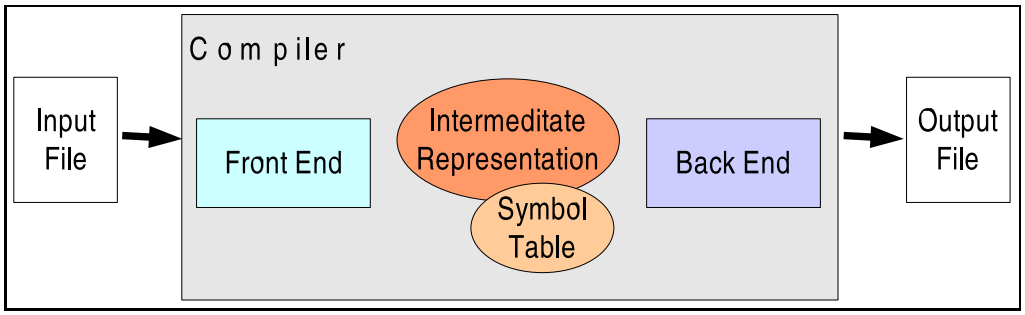


Figure 5.26: Common Compiler Structure

By using this structure, the problem of translating a certain information from a source language to a target language is mostly independent of the two languages. For an existing compiler a new back end for a compiler could be constructed which maps to another target language and the existing back end would simply be replaced by this new back end. As long as the two component fit together with the same intermediate representation, this replacement will work. The same holds for the front end as long as the new source language contains all information that is needed for building up the intermediate representation. This fact is shown in figure 5.27.

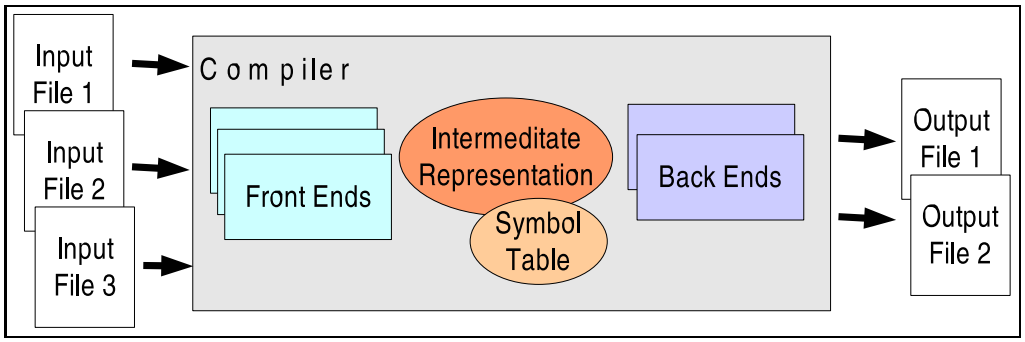


Figure 5.27: Common Compiler Structure using several front and back ends

As a prerequisite the content written in the source language has to be syntactically legal and depending on this language follow certain semantic rules that have to be well-defined in order to check whether this input file contains valid input for the compiler. If the compiler works well and the input file fulfills both properties, it will produce an output file that is syntactically and semantically legal in the target language. A compiler usually does not make any statements about how useful an input is or whether it makes sense in a certain way, as long as it correct in both point, syntax and semantics.

The compiler developed in this diploma projects follows this overall structure. It has a front end for a source language that was created during this thesis that was designed for specifying types

of zero-knowledge proof-of-knowledge protocols . The rules whether an input is syntactically and semantically legal is specified more in section 3. This compiler contains already two different back ends, one that uses LaTeX source code as target language and a second one that generates java source code.

Depending on the type of compiler and the goals it wants to achieve, different passes might be processed in the middle of this process. These passes just iterate over the intermediate representation and are therefore completely independent of the different front ends and back ends.

Now I will go on talking about the different parts that are involved in the order in which they are invoked by the process of compilation of an input file.

5.2 Front End

The front end of this compiler is the part that deviates most from the standard model of a compiler as explained in 5.1.

There are several reasons for this fact. The first point is that the user of this compiler does not have to specify the protocol that he wants to use in order to prove a certain secret by listing every single step of the computation. The user operates on the generalized protocol that is explained in section 2. The user simply has to specify all parameters needed in order to decide which instance of this generalized protocol should be generated instead of specifying the whole protocol on his own. In order to provide this service, the generalized protocol has to be stored somewhere within the compiler. Therefore the front end is extended by a library that contains all the information about the basic structure of the generalized protocol.

A second reason for extending the front end is the possibility to choose between two levels of abstraction or of how detailed the generated target code should be. This option makes sense only for a certain class of possible back ends but in these cases it has a strong influence upon the structure of the intermediate represenation that is built up by the front end.

The possibility to choose between a high level overview of the protocol structure and a detailed view that shows each single operation has to be mapped into the terms of software engineering. This could be done using two independent ways for building up the intermediate representation. This would require two different ways in the same front end. Both elements would need knowledge about the overall structure of the protocol and a lot of information would be redundant in these two parts which causes a lot of dependencies and has a very high impact for example on the maintenance costs of this software. Therefore we have chosen a different way of solving this problem.

Our solution provides a two step approach. In a first step, an intermediate representation is built that provides this highlevel view that hides lots of details about the number and structure of involved homomorphims and groups. This first step is performed by a component that is called the compositor. Now there is a second, optional step that iterates over this existing intermediate representation an builds up a new one, which is detailed unto the last operation that has to be performed. This second step is done using a component called expander that belongs to this extended front end too. If a user wants to receive this abstraction, the high level view on the protocol, the second step of expanding the compact intermediate representation is simply left out.

One could discuss whether or not the expander still belongs to the front end or if it is already one of the passes that is processed on the intermediate representation as mentioned before. I have decided to take the expander as part of the front end too as the compositor and the expander fit together very closely.

Now I will explain the different components that are involved in this extended front end more in detail.

5.2.1 Scanner and Parser

The scanner and the parser are the two parts that are mandatory for a compiler in general. This is the point where the knowledge about the syntax of the source language enters. The scanner is the part that has all information about the involved language elements - the keywords[10] - of the source language. The scanner reads the input file which is just a stream of characters in his view. It compares the actual characters according to some rules with the keywords of the language. As output it generates a stream of tokens. These tokens are all elements of the input language that are of any interest for the process of compilation. In this step all elements without any relevance are removed. In our case for example blanks or carriage return characters are of no interest and therefore are wiped out by the scanner.

Figure 5.28 shows a short section from the list of tokens for our compiler.

```
< KW_GROUP : "Group">
|
< KW_GROUPELEMENT : "GroupElement">
|
< KW_HOMOMORPHISM : "Homomorphism">
|
< KW_ADD : "+">
|
< KW_MUL : "*">
|
< IDENT : < LETTER > (< LETTER > | < NUM > | "_") *>
|
< LETTER : (["A" - "Z"] |["a" - "z"])>
|
< NUM : (["0" - "9"]) +>
```

Figure 5.28: Some tokens as specified in the input file for the compiler compiler

This stream of tokens is passed to the parser. The parser knows about the syntax of the source language. The process of parsing checks whether the input file is syntactically correct according to the syntax rules. As a second effect, actions can be specified that should be taken in case of a certain syntactic structure is recognized in the input file.
The parser works as a finite state machine that decides depending on the actual state and the following input symbols, which transition should be taken. To derive the structure of this finite state machine and to minimize it depending on the set of syntactic rules is quite a crucial and time consuming piece of work that can be automated using compiler compiler tools. I have chosen to work with a tool called javacc.[11] It is a scanner/parser generator for java that stands under the

[10] I will use the term 'keyword' for reserved words as well as for symbols for example of punctuation that have a certain meaning in the source language.

[11] This tool can be found at the URL https://javacc.dev.java.net/.

Berkeley Software Distribution (BSD) License and that merges both parts, the scanner and the parser, into one single component. It allows to annotate each syntax rule at an arbitrary point in the parsing process with arbitrary java source code. This gives a huge flexibility in extracting the information from the input file into some data structure that will be used by a java program later.

In a first step an input file for this compiler compiler has to be written that contains the grammar of the source language. This syntax is specified in terms of EBNF, a syntactic meta-language that covers the class of context free grammars. The standard EBNF is adapted in some points for reasons of convenience in order to generate a java class from it that does the parsing of the input file. Figure 5.29 shows such an EBNF production and the actions that are taken if this rule matches the input structure.

```
void IndexSet() : {

    // here is the declaration of local variables
    Token varT;
}
{
    < KW_INDEXSET > < EQUAL > < LBRACE > varT = < IDENT >

        {
            // at an arbitrary point in the production
            // actions in terms of java statements may be included

            ip.addVariableToIndexSet(varT.toString());
        }

    (< COMMA > varT = < IDENT >
        {
            ip.addVariableToIndexSet(varT.toString());
        }
    ) * < RBRACE > < SEMICOLON >
}
```

Figure 5.29: Rule from the scanner/parser input file

The parser in our compiler will raise a ParseException if the input file does not correspond to the syntax he is given to. Here is an example of such an exception:

```
Error: Encountered "{" at line 33, column 33.

Was expecting one of:
        ")" ...
        "||" ...
        "&&" ...
```

Some simple semantic errors such as if a certain symbol used in the input file was not declared before are found by the parser himself too. All other checks for semantic reasons have to be done later, as the compiler has not yet all informations needed for these checks as it is passing the token stream.

The parser fills in all the information into a data container class called `InputData`. All involved elements and all parameter choices are stored there. At the end of the parsing process, the input data class checks whether it is in a consistent state, for example if each group element is assigned to a group. If it is not, an exception will be thrown by the input data class.

After this check the first steps towards generating the protocol corresponding to the user's input is done and all user data is now consistent so far and stored in the input data class.

5.2.2 Library

In order to generate the intermediate representation of the protocol instance, we need to know what the generalized protocol structure looks like. This is stored in the library class. This class provides a method per round for each type of protocol, the Σ and the 2Σ protocol. Each method builds up a branch of the whole tree that contains the nodes that represent the different kind of operations that are processed in this round of the protocol. I will talk about this data structure and the different node types in detail later in this section.

As most rounds end up with a send message, an additional method per round is provided which returns a send node with all the appropriate data.

The whole library is designed as a static class with only static methods, as this library is stateless. If once a new type of zero-knowledge proof-of-knowledge protocols should be included into the compiler, the library would have to be adapted with the methods generating the node structure for this new protocol type.

The different methods of the library are invoked by the compositor that puts together the branches of the protocol in order to create one tree that contains the whole protocol structure — the intermediate representation of this protocol instance.

5.2.3 Compositor and Expander

After parsing all user input is stored in the input data container. Together with the basic protocol structure from the library, the structure for the desired protocol instance can now be composited. This composition process is done by the compositor class. It links the node structures it receives from the different methods in the library according to the chosen protocol type and depending on some additional facts for example whether several homomorphisms are involved in the protocol and therefore a secret sharing scheme is needed.

This process could be done just in one single step. We have chosen to split the composition of the intermediate representation into two steps. The main reason is that this choice gives the opportunity to provide the intermediate represenation in two different levels of granularity. Looking at the first prototype examples of protocol instances that had a certain complexity regarding the number and the structure of the involved homomorphism, we saw that the results were as detailed, the whole protocol as long that one could hardly get an overview of the overall structure. Therefore we decided to provide for the latex handout this choice between a mode that shows the overall structure but hides a lot about the complexity and a mode that shows each single operation.

We have discovered that the process of generating the detailed view, that we will call the verbose mode from now on, can be split up in two steps such that the abstract view called the compact mode is just the intermediate result of the first step. Here is an assignment in compact notation:

$$s_{\zeta_i} := r_{\zeta_i} + (-x_{\zeta_i}) \cdot c_i$$

Figure 5.30 shows the corresponding part from the intermediate representation.
In order to get the detailed view from the abstract view, one needs all information about the involved algebraic elements such as homomorphisms and groups, but no further information from

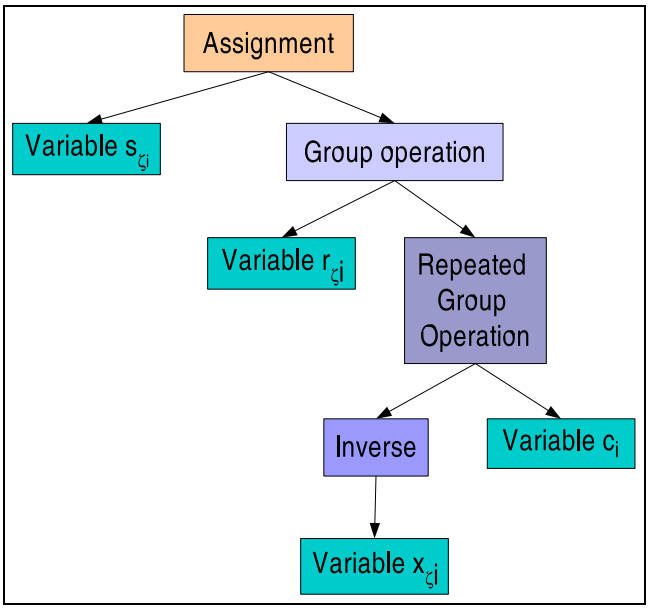


Figure 5.30: Node Structure of the Assignment in compact Notation as created by the Composer

the library will be used for this second step. So there is a clear separation between the part of the work that depends on the library and the protocol types and the other part that depends only on the structure of the algebraic elements.

Here is the same assignment as above now in verbose notation:

$$(s_1, s_2) := (r_1, r_2) + (-(x_1, x_2 + \tfrac{b+a}{2})) \cdot c_2$$

Figure 5.31 shows the corresponding part from the intermediate representation.
The separation of these two parts in generating the intermediate representation improves the design of the front end as it restricts the influence of the library and therefore the influence of potential changes regarding new or modified protocol types to a small well-defined area. This is once more an important point regarding the maintenance of the code.

The expander is a very interesting object from the point of view of software design patterns. It is the first of six different classes in the compiler implementation that implement a sort of visitor pattern as explained in [4]. Each operation on the whole intermediate representation data structure is implemented as a visitor pattern. The visitor pattern is one of the behavioral software design patterns. It is used to implement an operation that involves objects in a complex structure. One way of implementing this would be to provide logic in each class that could occur in the structure to support the operation. The visitor pattern provides an alternative way to implement such operations that avoids complicating the classes of the objects in the structure by putting all of the necessary logic in a separate visitor class.

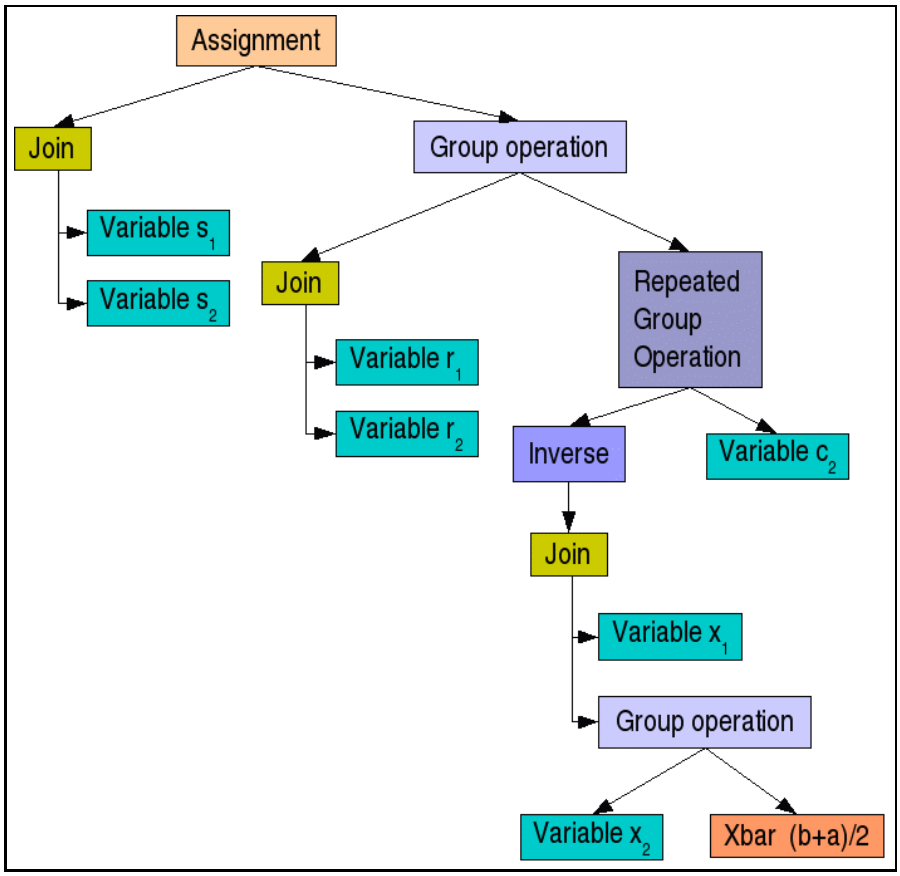


Figure 5.31: Node Structure of the Assignment in verbose Mode as created by the Composer and expanded by the Expander

The visitor pattern allows the logic to be varied by using different visitor classes. In our case we will have to visit each node in the intermediate representation not only once but several times always performing different operations on the data structure. Therefore we have different visitor classes, each of them with just one single task.

The visitor pattern as it was introduced is intended to act on every single object of the data structure but not to change the data structure itself. It is evident that it is rather a challenging task to modify a data structure while you are operating on it without cutting off one's own branch. In order to append to a certain object a new node structure that was generated by a successor of this object, the function "visit" that traverses the tree gives a return value, which is different from the design pattern as it is usually found in literature. This value returned from the successor's visit can be appended at the object itself.

An additional difficulty is the problem that some parts of the structure are visited several times, e.g. for each involved homomorphism once. This part of the structure may not be modified, but the visitor has to generate a copy of it. Otherwise some homomorphisms would generate wrong code as they would expand a structure that was already expanded before.

The expander class is a very powerful and probably one of the most interesting classes, but also one of the most delicate points in the whole project regarding future changes. Therefore I have applied some techniques known as "Design by Contract" using preconditions to the methods of this class as described in [6].

5.3 Intermediate Representation

The intermediate representation is the core of a compiler. The front ends and the back ends might change but the intermediate representation stays the same independent of the source and target language. The intermediate representation mirrors the input with all information which could be relevant for the translation.

Therefore the intermediate representation is the point where the design of a compiler according to my opinion should start. In my diploma thesis the intermediate representation was the first part that was developed at the very beginning.

In order to devise an appropriate representation of an arbitrary problem, which is always an abstraction, the problem itself has to be understood in depth. So my diploma thesis started with the study of the different protocol instances with their facets and the different ways they could be combined with each other. During the diploma thesis it turned out that the different protocols could in fact be considered as different instances of one general protocol.

The intermediate representation is based on the different operations that occur in a zero-knowledge proof-of-knowledge protocol , e.g. group operations, assignments, comparisons etc.. The intermediate representation is a kind of tree data structure with all operations represented by the appropriate nodes. The design of the node hierarchy consisting of all different node types and their hierarchy has to satisfy some general rules. The set should be minimal in the sense that, if two node types are very similar in some way, they should be taken as two occurences of the same node type maybe with some additional flags in order to model the differences. The number of different nodes should be as small as possible because each different node type will result in a method of its own in all visitor patterns throughout the whole compiler. On the other hand, if the number of different node types is chosen too small this will end up in huge "visit"-methods with lots of branching instructions. [12]

If not only the number of nodes is chosen in a reasonable way but also a type hierarchy is used to express the dependecies between the node types, the implementation of the visitor classes will once more be simplified. I will present the node hierarchy I have chosen for my diploma thesis more or less in detail in section 5.3.1.

[12] In fact that is exactly the point the visitor pattern wants to get rid of.

5.3.1 Node Type Hierarchy

The intermediate representation depends strongly on the node types and their internal hierarchy. Therefore I want to explain this structure and the reasons for choosing this design in an own subsection.

All node types implement the general node interface called `NodeInterface` within the core package. This interface provides the basic methods that can be used for all different kinds of nodes throughout the whole hierarchy. There are the methods `accept` for both two types of visitor classes — those from the expander with a return value and the usual ones without return value. Furthermore there are some access methods that are valid for all kind of nodes such as the `public NodeInterface getNext()` and `public void setNext(NodeInterface n)` and for several flags.

The parent class of all node types is the class `Node` that gives a default implementation of the methods postulated by the interface. Now there are four categories of subtypes as shown in figure 5.32:

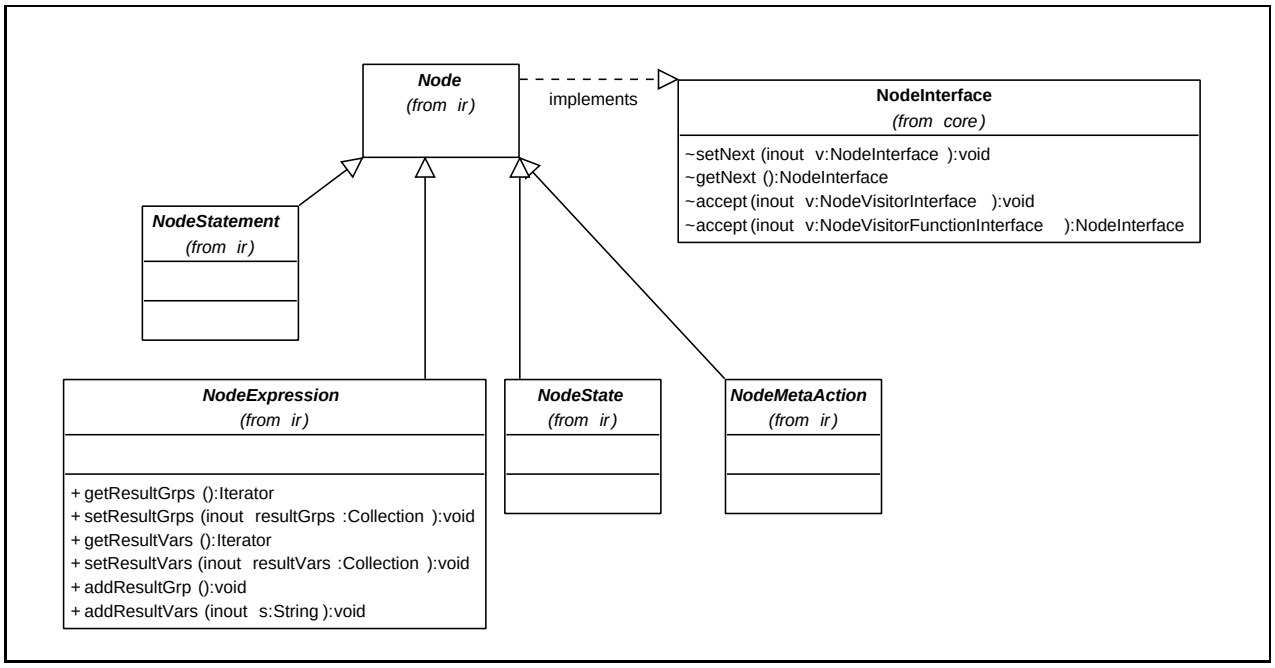


Figure 5.32: Subtype Categories from Node

Here is a short description of the main function of the nodes in the four categories each with a UML class diagram that shows the most important operations.

- The first categories are those nodes that represent a statement such as a comparison or an assignment. These are the nodes that line up in the chain of actions that follows directly for example from the round node. The hierarchy of this node category is shown in figure 5.33. I want to mention just some of the node classes whose function is not that obvious as for example an assignment node.

  NodeRandomChoice This node contains the information which variables should be chosen equally distributed at random. In case where there are constraints on preimages, it chooses at random for those variables which are independent and creates the subtree for calculating the dependent ones during the expansion phase.

  NodeIntervalCheck, NodeConstraintsCheck In case that such additional features are chosen, these checks are performed at that place in the protocol where these node can be found in the intermediate representation. The additional information for example in which interval the variable should be or what the constraint is can be found in the symbol table.

  NodeGenerateG_i, NodeGenerateRho, NodeGenerateSRSA If preimages are included in the index set, the 2Σ protocol is applied. In this case additional actions have to be performed, that are modelled by these three node types.

  NodeGenerateCommitRandomness This node class is used only for the 2Σ protocol. A new node type is necessary because at compile time one has no idea about what type of commitment scheme will be chosen for execution and therefore what type of randomness will be necessary. This is the reason for not using the usual NodeRandomChoice because that is just the choice of a group element from a certain group or an integer interval.

- The second category contains nodes that represent an expression. These nodes represent for example the left and the right hand expressions of an assignment. They may not stand alone but occur only as predecessor of a node from the first category that represents a statement. The subclass "NodeOperation" itself is a superclass for all those expression nodes where an operation needs a well defined operation sign for the latex notation. The function of the node classes as shown in figure 5.34 is mostly clear by their class names except for the NodeXBar. The NodeXBar is a representant for the shift operation in case of infinite groups. For reasons of performance the variables are shifted by the middle of the interval where the preimage is chosen from. As this is just always the same calculation I have decided to create this node class as a kind of shortcut instead of modelling the whole calculation $\frac{a+b}{2}$ each time by the whole subtree of operations.

- The third category consists of nodes that represent an internal state such as one specific homomorphism on which the operations are performed. All nodes following this node act only in this state, i.e. upon this homomorphism specified by the internal state node. The NodeRound is necessary because some operations are different whether they are processed by the prover or the verifier, simply as the code generator for example in case of java source code as target language needs to know in which class he has to write. In case of several homomorphisms the action depends on whether a prover knows a certain preimage or not.

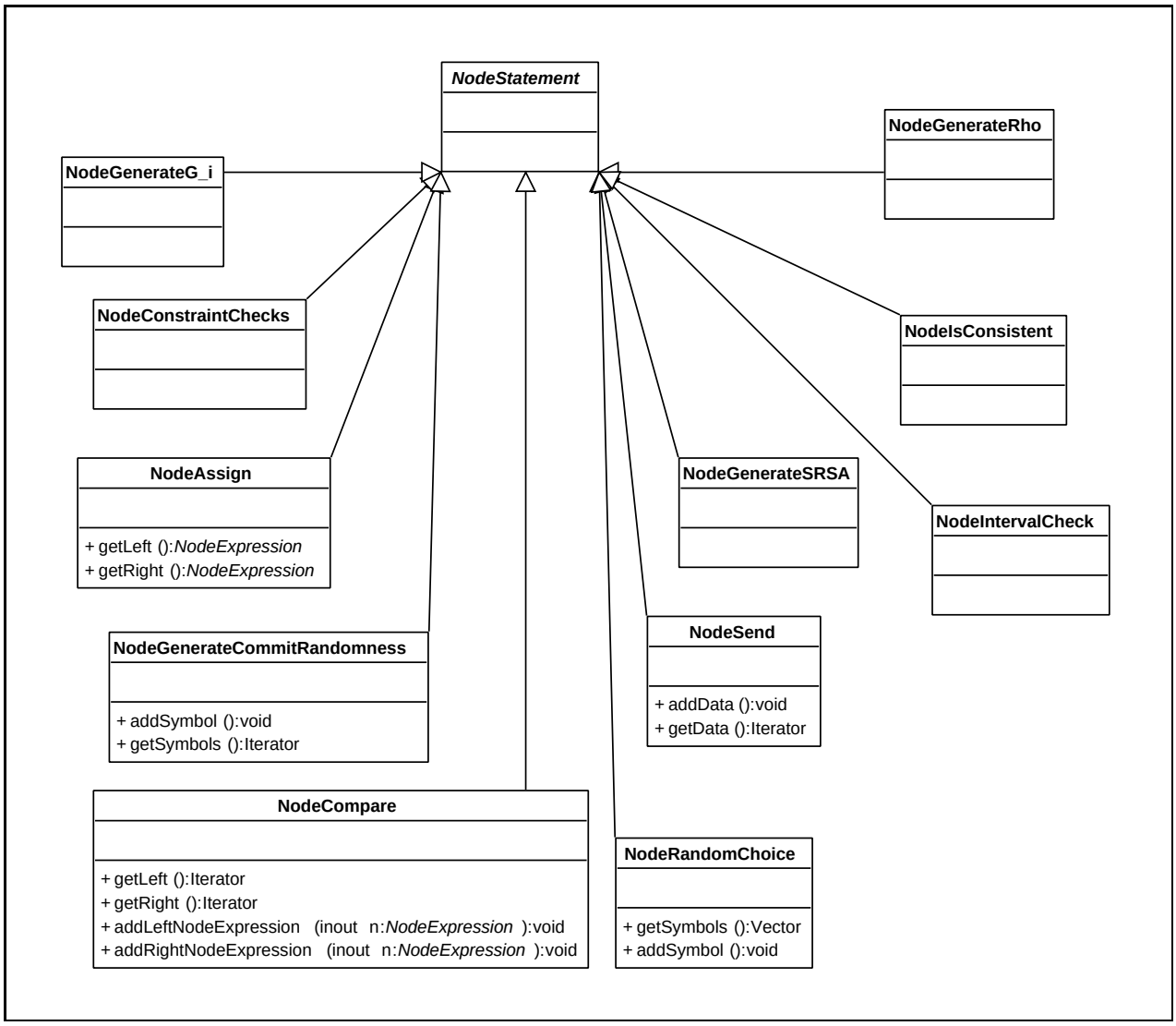


Figure 5.33: Class diagram of category statement nodes

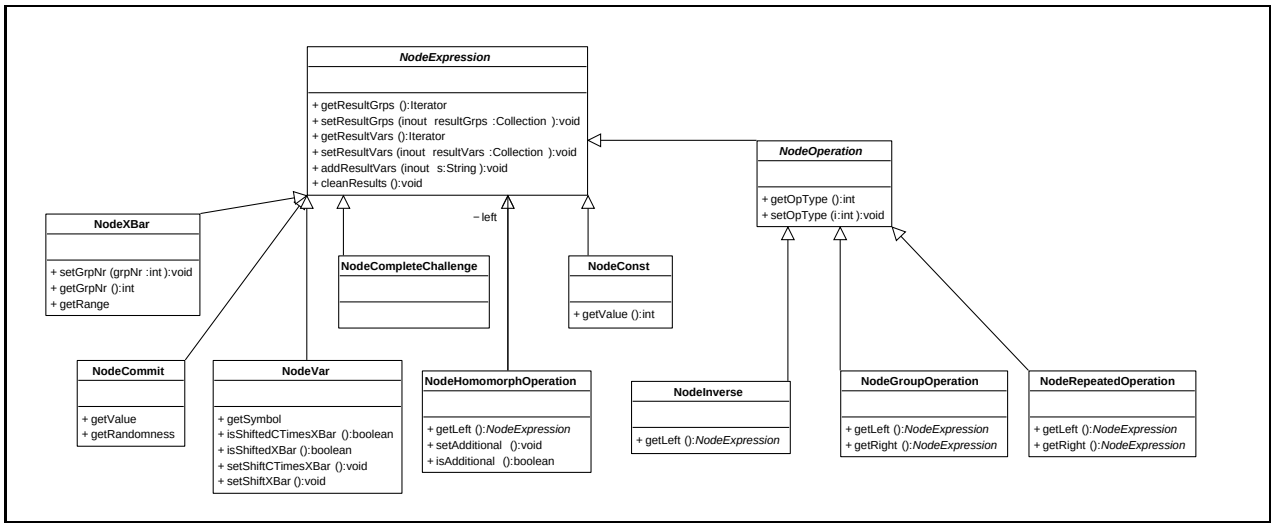


Figure 5.34: Class diagram of category expression nodes

Therefore there are three children of the NodeDifferentBehaviour. The two subclasses that extend this class is one for the compact mode where everything is written just as an iteration over all homomorphisms - the NodeForEach - whereas the NodeIndependentHomomorphism is used for the verbose mode and appears in each round as often as the number of homomorphisms. Figure 5.35 shows this type hierarchy.

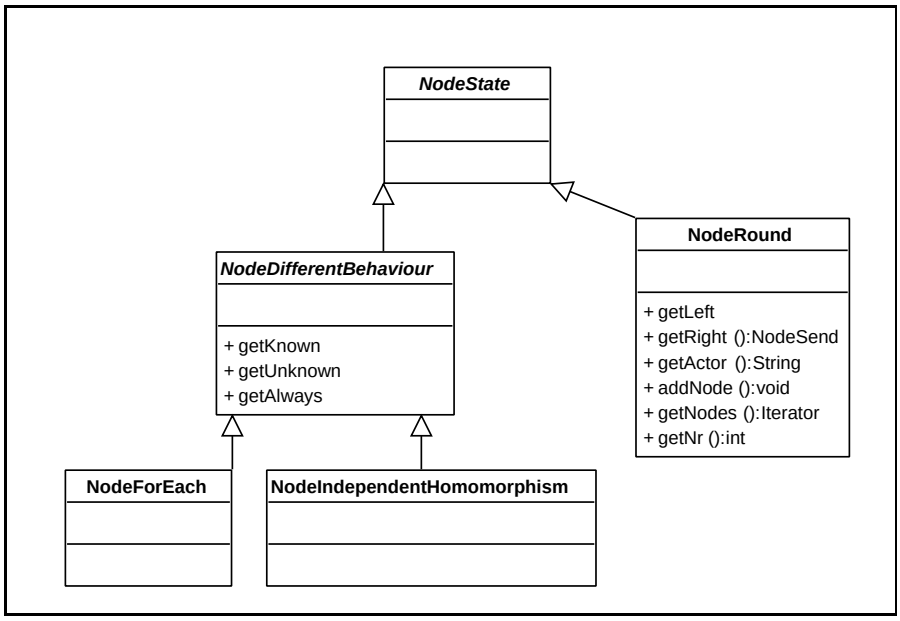


Figure 5.35: Class diagram of category state nodes

- The last category are those nodes that represent an action in a sense of meta level. This is for example the node that indicates that the additional homomorphisms for the 2Σ Protocol should be instantiated at a certain point in time. This node will appear twice in the tree because the prover as well as the verifier needs the additional homomorphisms. The second node class in this category is needed for the 2Σ protocol because the computations for the values starting at round 4 do not have to be processed within their own groups, but in the additional infinite group. Therefore at this point in the protocol, the variables have to change their corresponding group as seen in the symbol table. These two node classes are shown in figure 5.36.

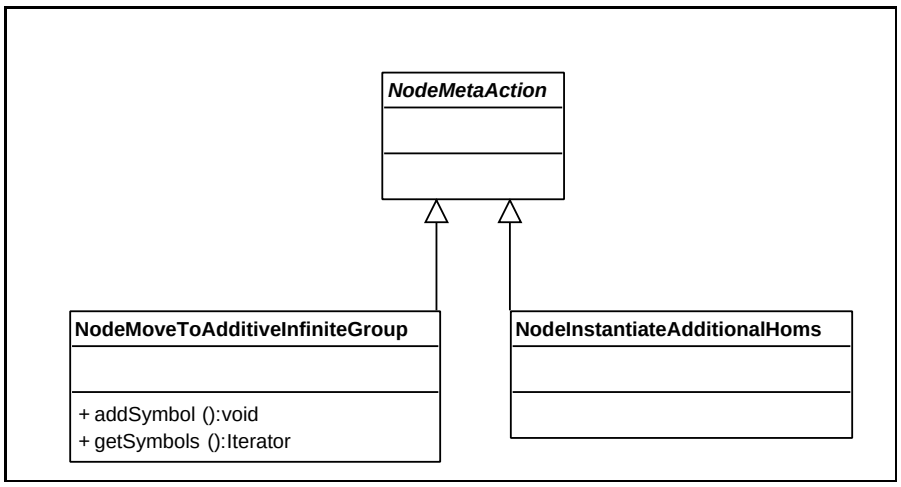


Figure 5.36: Class diagram of category meta-action nodes

5.3.2 Structure

The intermediate representation uses the following structure: The different branches are connected through a chain of the round nodes. Each round node stands for one protocol round. It contains the number of the round, the actor - whether the prover or the verifier - and two links to the actions that take place during these round. The first link leads to the branch where all computations, e.g. assignments or comparisons are within, the second link leads to the send node that contains the data that is transmitted at the end of this round if there is a transmission at all. Figure 5.37 shows this overall structure.

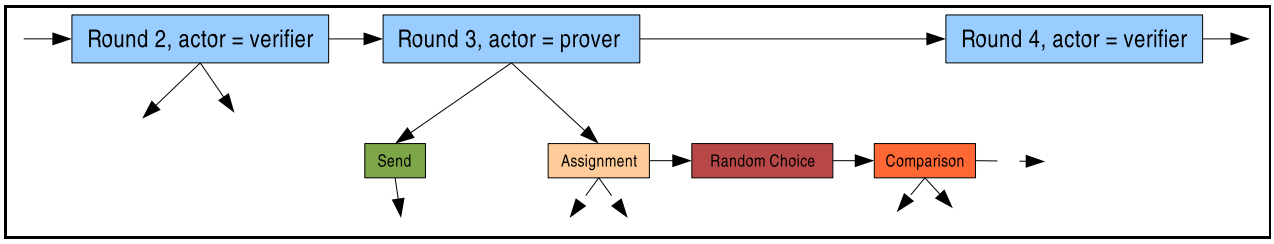


Figure 5.37: Overall Structure of the Intermediate Representation

The minimal size of an intermediate representation of the most simple instance of a zero-knowledge proof-of-knowledge protocol from our generalized protocol consists of about 40 nodes. As soon as several homomorphisms are involved that have each of them a structure which is a little bit more complex and some additional features as constraint checks and preimages which are included in the index set are chosen, the size reaches thousand nodes pretty soon.

For debugging reasons it is important that this structure can be visualized in some way. Therefore a trace output that shows the intermediate representation either after each visitor traversal or at least at the end of the process is provided. This trace output is only text based and the dependences are shown by using different indents. A section of this output that can be turned on or off simply by flipping a boolean flag in the main class is shown in figure 5.38.

```
- - - - - - - - - - - - - - - - - -
 [NOD_ROUND]( nr = 4, actor = Verifier)
- - - - - - - - - - - - - - - - - -
  [NodeCompare]
    left  =
      [NodeJoin] (resultVar = [t_0_0], resultGrps = [(H_1 *)])
        [NodeVariable](symbol = t_0_0, resultGrps = [(H_1 *)])
    right  =
      [NodeGroupOperation] (resultVar = [var10], resultGrps = [(H_1 *)])
        left=
          [NodeHomomorphOperation] (resultVar = [var8], resultGrps = [(H_1 *)])
            left=
              [NodeJoin] (resultVar = [s_0_0], resultGrps = [(G_1 +)])
                [NodeVariable](symbol = s_0_0, resultGrps = [(G_1 +)])
            right=
              [NodeJoin] (resultVar = [var9], resultGrps =  [(H_1 *)])
                [NodeRepeatedOp] (resultVar = [var9], resultGrps = [(H_1 *)])
                  left=
                    [NodeVariable](symbol = y, resultGrps = [(H_1 *)])
                  right=
                    [NodeVariable](symbol = c, resultGrps = [(Z +)]
```

Figure 5.38: Section from the trace output of the intermediate representation

5.4 SymbolTable

As the intermediate representation is the heart of the actions that take place while the protocol instance is executed, the symboltable is the place where all information about the involved elements can be found. The symboltable has to be built just like the intermediate representation by the front end of the compiler.

At the beginning of the compilation process — during the scanning and parsing phase — most information is just stored in the input data class in form of simple strings. For later generation of target code, it is much more convenient to work with more complex object that mirror the algebraic elements that are involved in the protocol. For this reason we build up the symboltable that consists of different types of symbols such as group symbols, variable symbols, homomorphism symbols and so on.

Each of these symbol types is modelled as a class of its own. Figure 5.39 shows the hierarchy of these symbol classes.

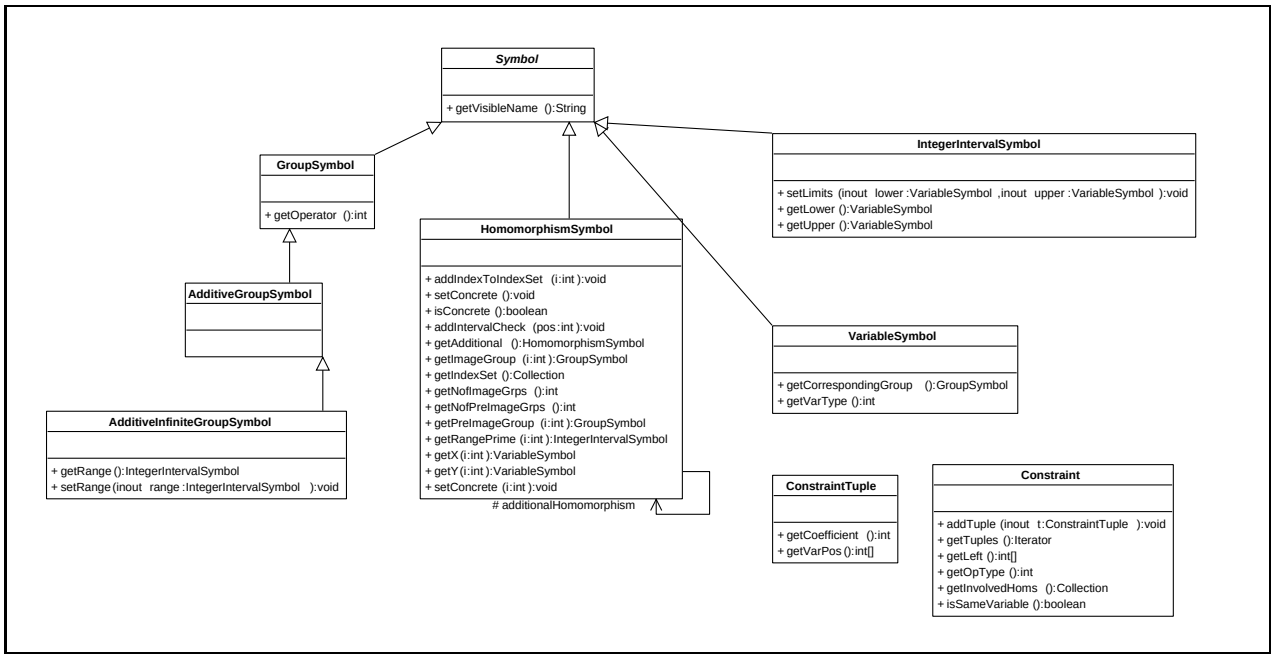


Figure 5.39: Hierarchy of the symbol classes

5.5 Semantic Completion and Analyze

When the front end has completed its work, the structure of the intermediate representation and the symbol table with all its element are fixed. We have chosen to guarantee for our compiler a kind of type safety. I will explain this with a simple example:

If we generate java code, at some point in time we will have a group operation on two group elements. The group operation is performed on the group that the elements belong to. This could look like this.

```
GroupElement a, b, c;
Group G;
... // these GroupElements are created somehow
c = G.operate(a,b);
```

It is evident that the implementation of the group operation can only work if both elements are from the same group and the group operation is really performed on that group. Type safety in our case means that we want to guarantee that no operation is performed to elements that can not match. Therefore we have rules on different operations such as group operations, repeated group operations where the second argument has to be of a type that can be turned into an integer, but also assignment and comparisons where both arguments have to be of the same order in case of tuples and of the same type.

This checking needs some precomputation. At the end of the front end's work, each variable and constant node in the intermediate representation knows of what type — now used as "from which group" — it is. But all operations that are at a higher point in the intermediate representation tree do not know anything about the types of their depending subtrees. We have to walk through the tree in a depth first traversal such that each node can get as much information about his children as possible. This is done by the SemanticCompletionVisitor, another application of the visitor pattern.

Figures 5.40 and 5.41 show how the intermediate representation before and after that step.

```
[NodeAssign]
  left=
    [NodeJoin] (resultGrps = [])
      [NOD_VAR](symbol = t_0_0, resultGrps = [])
  right=
    [NodeHomomorphOperation] (resultGrps = [])
      left=
        [NodeJoin] (resultGrps = [])
          [NOD_VAR](symbol = r_0_0, resultGrps = [])
          [NOD_VAR](symbol = r_0_1, resultGrps = [])
```

Figure 5.40: Section from the trace output before semantic completion

```
[NodeAssign]
  left=
    [NodeJoin] (resultGrps = [(H_1 * )])
      [NOD_VAR](symbol = t_0_0, resultGrps = [(H_1 * )])
  right=
    [NodeHomomorphOperation] (resultGrps = [(H_1 * )])
      left=
        [NodeJoin] (resultGrps = [(G_1 + ), (G_2 + )])
          [NOD_VAR](symbol = r_0_0, resultGrps = [(G_1 + )])
          [NOD_VAR](symbol = r_0_1, resultGrps = [(G_2 + )])
```

Figure 5.41: Section from the trace output after semantic completion

After the semantic completion, these type checks can be done. It is once more a visitor who will operate on the whole intermediate representation and check whether this type safety is given. This decision is made always regarding the actual node and its children if they exist according to the set of rules the visitor has for each node type. If one of these rules are violated, an exception will be thrown.

An important remark is that such a semantic analyzer exception should not be thrown by the compiler after it has been released. This exception can not be brought about by the end user because the intermediate representation structure is generated by the compositor using the library class and the expander. So if such an exception is thrown, something with the generation of the tree must be wrong. I have left this part of type checking which was very useful for developing the compiler in the source code, because the whole project is still quite far away from a stable release and it is very likely that things will be added or maybe changed again. Therefore this tool is still active.

The whole process of semantic completion and semantic analyze only makes sense on the expanded intermediate representation. In the compact form of the intermediate representation the variables are not real groups and group elements but only representants of a set of variables. Therefore they all have a variable as index.

5.6 Back End

Now we will look at the back end. This is the part where the target code is generated. As the purposes of this diploma thesis were to generate on one hand a protocol specification and on the other hand executable code for testing, two independent back ends are completed. Each backend depends only on the intermediate represenation and the symboltable as it was generated by the front end with the additional completion that was done by the SemanticCompletionVisitor.

Figure 5.42 shows the common interface for the back ends.

core

Interface CodeGeneratorInterface

All Known Implementing Classes:
CodeGeneratorJava, CodeGeneratorLatex

---

public interface **CodeGeneratorInterface**

Interface for the backend of the compiler. Generates the target code based on the intermediate representation and the symbol table.

Author:
Thomas Briner
Nov 21, 2003

Method Summary

void	**generate**(NodeInterface intermediateRepresentation, SymbolTableInterface SymbolTable, java.io.Writer[] wrt) Generates the output code and writes it into the Writer wrt

Figure 5.42: Javadoc of the CodeGenerator Interface

5.6.1 Java Code Generator

The goal of the java back end is to produce runnable code such that a prover and a verifier instance could run the protocol. Each of them will need some input as it is specified by the protocol specification which is given in the constructor of the class. At the moment when a user compiles an input file in order to produce this java code for testing, the concrete implementations of the homomorphisms for example might not be known yet. So the generated code has to work on an abstraction level that is provided by the java language through the mechanism of interface design.

Before writing the code for the java back end, this interface design has to be done. There have been already other implementations with a corresponding interface design from student projects in connection with the idemix system that was mentioned in section 1.2. As these implementations were very much focussed only on some special groups they could not be used as a whole for my diploma project. Some group implementations I have written for my diploma thesis use the work from Dieter Sommer as calculation unit but all interface design and most implementations for prototype testing had to be done by myself. The java interface design as it is used in this diploma project is described in section 6.

The output consists of the two classes, one for the prover and the verifier and additional classes for the parameter passing as the messages are sent. If a homomorphism for the protocol instance is specified as a concrete homomorphims, additional classes that implement this homomorphic operation are generated. Further details about these generated files can be found in section 4.2.

The java code generator uses as input the intermediate representation the symboltable and the output streams it should write to. It is implemenented — of course — as a visitor pattern. Depending on whether the actor is the prover or the verifier as specified in the node of type round that sets the state, it writes to one or the other file.

The code generator has a preliminary section that writes the import sections, the variable declarations and the constructor. The constructor arguments depend completely on the specified protocol, whether the homomorphisms are needed as arguments or they are already specified as concrete functions in the input file, whether a secret sharing scheme is needed and so on. The constructor parameters correspond directly with the items listed in the LaTeX output under the sections common and preimage input.

If the protocol instance consists of several homomorphisms it might be the case that the prover does not need to know all preimages but just a subset that corresponds to the access structure. In this case the constructor asks nevertheless for all preimages but the program invoking the constructor simply gives `null` as input for all those preimages that are unknown. Out of these arguments, the prover class figures out the subset of preimages that is known.

In the constructor there are some precomputations for example for the integer interval limits that might be used in the program several times. This is done simply for performance reasons.

The code generated by the compiler is not optimated e.g. there are several assignments that may be optimized as two assignments could be merged into one. But as the time-consuming operations are all those where group operations are involved, we did not wanted to spend too much time in optimizing points that do influence the overall running time only in a negligible way.

There are two more classes that are involved in the generation of the java source code that I want to mention for reasons of completeness in the following two paragraphs.

Parameter Class Generator To guarantee a low level of type safety at compile time, I decided to exchange the parameters from one actor to the other not simply by storing them into some kind of container where wrong types would raise ClassCast exceptions only at runtime. Therefore a class of its own for each send message is created that contains exactly those variables that have to be exchanged for this special instance specified by the user. The class consists only of all data stored in instance variables and getters and setters for all those fields. The generation of these parameter classes was automatized by a class called `ParameterClassGenerator` in package `util` in the source code. It contains the static method that returns a string that can be written to a file stream and that contains the parameter class for this round. It takes as input the round number that is used in the class name and a collection of all variable symbols to be sent.

Concrete Homomorphism Generator If a user specifies a homomorphism in the input file as a concrete function, this homomorphism is generated directly as a concrete implementation of this function that may be an arbitraty expression in terms of group operations as long as the homomorphic property as defined in section 1.1 is given. This homorphism implementation is generated as a class of its own who's name is given by the name of the homomorphism as specified in the input file. According to the interface design described in section 6 the method `public Iterator image(Iterator it)` must be implemented. The generation is automatized by another helper class from package `util` called `ConcreteHomomorphismGenerator`. It takes as input the homomorphism symbol and returns the implementation of the homomorphism as a string so that it can be written into the file stream.

I will tell more about this generation of the homomorphism implementation class from an arbitrary homomorphic function and the way this function is stored in the homomorphism symbol in section 5.7 where I will give a short sketch of some problems of implementation and their solution.

5.6.2 Latex Code Generator

The latex code generator is the second implementation of the back end that is done in my diploma thesis. It is based exactly on the samemechanisms as the java code generator. It works as a visitor pattern as well and takes as input the intermediate representation, the symbol table and the stream to write to.

The first part of the code generator uses only the symbol table in order to produce the declaration section of the input file where all involved elements are mentioned. It lists the homomorphisms and - if infinite groups are involved - the function that is used for the computation which is no longer a homomorphic function strictly according to the definition. It shows the common and the private input and depending on the input additional facts such as the list of constraints or interval checks or the access structure. All these information can be extracted from the symbol table without much precomputation.

The second part where the protocol execution is written down is generated by the node visitor that iterates over the intermediate representation. The fact whether the intermediate representation was expanded or not does not influence the generation of the latex code at all — except for the title of the second part.

Because of the fact that most variable names can be chosen by the user and some additional requirements such as the one that a variable called `sigma` should be displayed as σ, the way how to format names in the latex code generator is not too obvious. I will say a few more words about this problem in section 5.7.

5.7 Some Remarks on the implementation

In this section I want to present some interesting points on the implementation level. They all have in common that some problem has to be solved who's connection to the compiler is not too obvious. The implementations of my solutions for this kind of problems are collected in the package util that provides a toolbox for problems of different types.

5.7.1 Handling the Access Structure

The user has two different ways how to specify the access structure as explained in section 3.3.1: He can either specify a set of qualified sets or he can give a boolean formula using the homomorphisms as litterals.

In the second case the compiler has to perform several steps in order to find the qualified sets which are needed in order to instantiate the implementation of the access structure.[13] The boolean formula given by the user is not an arbitrary boolean formula, it follows certain rules. As a restriction no negation is allowed at all. That means that no negated litterals are accepted either. Therefore the set of qualified sets can be extracted out of formula the user specifies using efficient algorithms.

The first point is to store the formula in a way that is convenient for later steps. I have decided to store it as a binary tree structure that consists of two node types. Each node is either a leaf with a single non-negated litteral stored in it or it is a boolean operation, either $\wedge$ (in java notation `&&`) or $\vee$ (in java notation `||`) with two child nodes.

Figure 5.43 is taken out of a possible input file.

```
Relation =
    ([(y_1) = zeta_1(x_1)] ||
    ([(y_2) = zeta_2(x_2)] && [(y_3) = zeta_3(x_3)]))
    && ([(y_2) = zeta_2(x_2)] || [(y_1) = zeta_1(x_1)]);
```

Figure 5.43: Specification of an access structure in the input file

The formula corresponding to this input could be written as

$$(\zeta_1 \vee (\zeta_2 \wedge \zeta_3)) \wedge (\zeta_2 \vee \zeta_1).$$

Figure 5.44 shows the same formula now as a binary tree as it is built up by the parser.

[13] If no access structure is defined in the input file, the access structure has to be given at runtime as an argument to the constructor in case of java source code generation. If an access structure is specified, the creation of this access structure has to be done behind the scenes by the compiler.

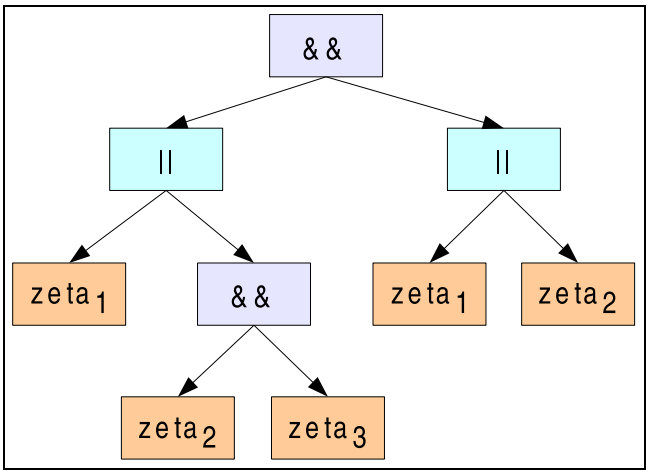


Figure 5.44: Same formula now written as tree as built by the parser.

In order to get the set of qualified sets, we need the formula in the disjunctive normal form (DNF). A boolean formula F is in DNF, if F is of the form

$$F_1 \vee F_2 \vee \cdots \vee F_k,$$

where F_i is a conjunctive term for any i.

Each formula can be tranformed into DNF by applying the distributive law. In our representation as a tree, the distributive law is shown in figure 5.45.

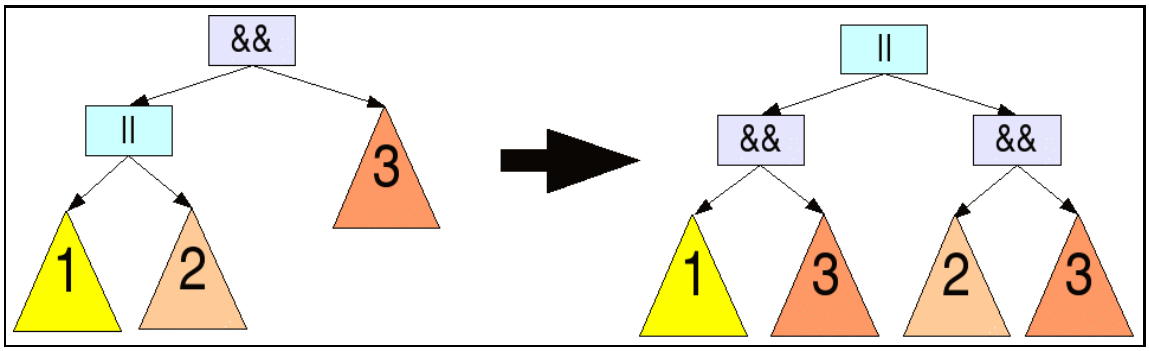


Figure 5.45: The distributive law applied to the tree representation of a boolean formula.

How can we decide whether a formula in the binary tree representation is already in DNF? The rule I figured out is like that: If at no point in the tree an operation node with the operation $\wedge$ has as a child an operation node with the operation $\vee$ in it, the tree is in DNF. This rule can be checked recursively for every node in the tree.

Now we have the tools to extract the DNF out of a boolean formula in tree notation. We take the input tree that is generated by the parser and apply the distributive law by traversing the tree

at any point where an operation $\wedge$ is the direct predecessor of an operation $\vee$. One single pass might not be enough as the $\wedge$ operation is pushed down only by one step. The upper bound for the number of iterations it could take is the height of the tree which is at most $n-1$ where n is the number of litterals. In the average case the result would be even better as the height of a tree is in $O(log(n))$ in the average case.

As a result we get a tree whose corresponding formula is in DNF. Figure 5.46 shows the tree from figure 5.44 after having applied the distributive law to all nodes where it is necessary.

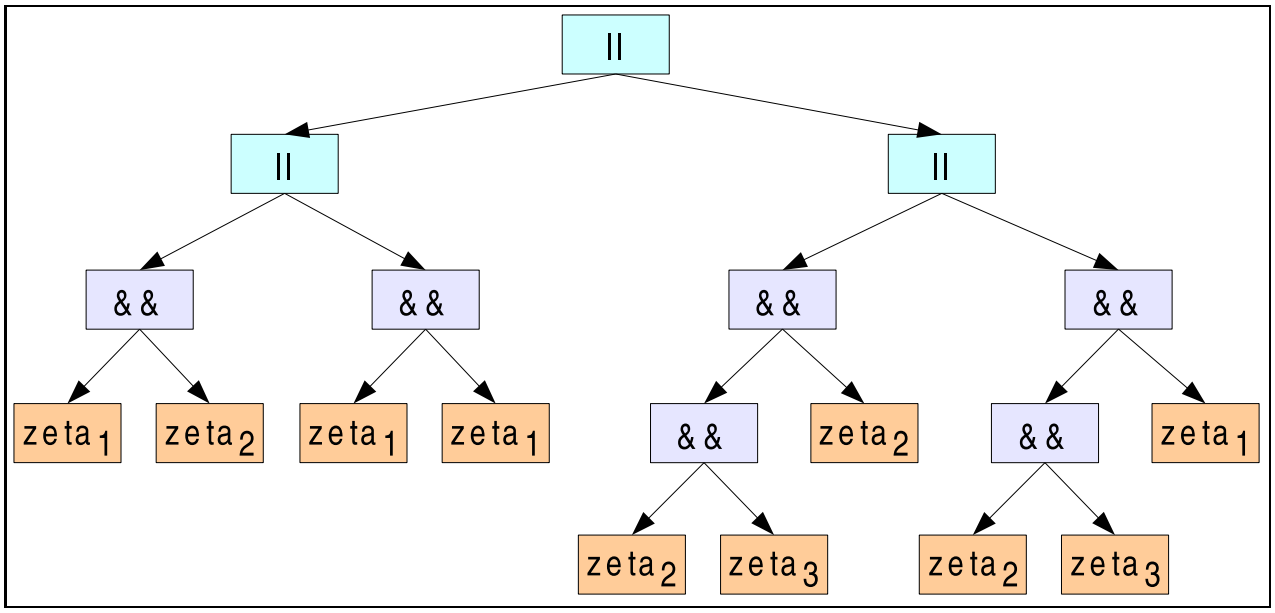


Figure 5.46: The binary representation of the formula after having applied the distributive law.

The corresponding boolean formula is

$$(\zeta_1 \wedge \zeta_2) \vee (\zeta_1 \wedge \zeta_1) \vee (\zeta_2 \wedge \zeta_3 \wedge \zeta_2) \vee (\zeta_1 \wedge \zeta_2 \wedge \zeta_3).$$

Now after eliminating the litterals that have several occurrences in the same conjunction, the formula is

$$(\zeta_1 \wedge \zeta_2) \vee (\zeta_1) \vee (\zeta_2 \wedge \zeta_3) \vee (\zeta_1 \wedge \zeta_2 \wedge \zeta_3).$$

This formula is in DNF, but still it does not directly show the qualified set for the access structure. The set of qualified sets has to be monoton. In other words, if a set $s, s = \{\zeta_1\}$ is already a qualified set, all supersets of s should have access too. So we have to minimize our sets that we get from the conjunctions of our DNF such that no set occurs that is a superset of another one.

That gives us from the initial set $t, t = \{\{\zeta_1, \zeta_2\}, \{\zeta_1\}, \{\zeta_2, \zeta_3\}, \{\zeta_1, \zeta_2, \zeta_3\}\}$ the minimal set $t', t' = \{\{\zeta_1\}, \{\zeta_2, \zeta_3\}\}$ which are the qualified sets from the access structure specified by the user.

All these operations are performed in a class called `BooleanFormulaToDNFConverter` which is located in package `util`.

5.7.2 Constraints on preimages and the Access Structure

It is possible to define constraints on preimages from the same group.[14] These constraints can be defined explicitly or implicitly as described in section 3.3.2.

Explicit Description Constraints that are defined explicitly are stored during parsing in objects from the class `Constraint` in package `symbols`. They consist of the left hand side variable and a collection of tuples, each another preimage and the corresponding coefficient.

Implicit Description If the same preimage occurs in several homomorphism, this is handled exactly the same as if the different occurrences were given different variable names and an explicit constraint would have been added that connects the preimages with a coefficient of one. A constraint object is created by the Compositor that does this checking for implicit constraints.

Now there might be a problem in defining constraints regarding the access structure. I want to illustrate this connection using an example.

Given the following input file:

```
// Declarations
Group (G_1,+);
Group (H_1,*);
GroupElement x_[1..3];
GroupElement y_[1..3];
Homomorphism zeta_[1..3];

// Assignments
AssignGroupMember(G_1, {x_[1..3]});
AssignGroupMember(H_1, {y_[1..3]});

// Definitions
DefineHomomorphism(zeta_1, G_1 -> H_1);
DefineHomomorphism(zeta_2, G_1 -> H_1);
DefineHomomorphism(zeta_2, G_1 -> H_1);

// Protocol
SpecifyProtocol [
  Relation [
    CommonInput = {(zeta_1, (y_1)),(zeta_2, (y_2)),(zeta_3, (y_3))};
    PreImageInput = {(x_1),(x_2),(x_3)};
    AccessStructure = {{zeta_1, zeta_2},{zeta_2, zeta_3}};
  ]
  Constraints = (x_1 = 3* x_2) && (x_3 = x_2);
  Target = LATEX;
```

[14] The same effect could be achieved by hiding this constraint behind the group interface. Every time a new element is chosen at random these constraints are considered behind the scenes. The disadvantage of this way of modelling constraints is that this is not visible in the protocol specification that is generated in LaTeX .

]

Now we want to guarantee that the constraints $x_1 = 3 \cdot x_2$ and $x_3 = x_2$ do hold. According to the access structure the proof should be accepted if we either know x_1 and x_2 or x_2 and x_3. In both cases we will not be able to guarantee both constraints, as there is always one constraint where one variable is involved that we do not have to know according to the access structure and for which the protocol will be simulated. Therefore we can not guarantee that the constraints hold.

A constraint that is not compatible with the access structure does not make sense and should therefore not be allowed. Now I have to define, what compatible precisely means.

Definition 1 *A set of constraints is compatible with an access structure defined by its qualified sets iff the following holds for each constraint and each qualified set: The set of involved homomorphisms of the constraint is either a subset of the set of homomorphisms included in the qualified set or the two sets are disjoint.*

This check has to be performed for all constraints — the explicitly defined ones as well as for the implicitly defined ones.

If no constraints are defined, the computations in the protocol can be performed for each homomorphism separately, modelling the access structure simply as an input to the secret sharing scheme. If there are constraints, the computations in one homomorphism might depend on computations (especially the random choices) of other homomorphisms. They have to be treated as one homomorphism and their computations have to be performed using the same share from the verifier's challenge.
Here is another example:

Figure 5.47 shows the homomorphisms as they were defined by the user in the input file.
The access structure, the constraints and the relations are as shown in figure 5.48.
First the compiler has to check whether the constraint is compatible with the access structure. It is, as the involved homomorphisms ζ_1 and ζ_2 are both included in the first qualified set and the intersection with the second qualified set is empty.

1.1 Homomorphisms as defined in Input File

Homomorphism ζ_1

$\zeta_1 : G_1^2 \to H,$

Homomorphism ζ_2

$\zeta_2 : G_1 \times G_2 \to H,$

Homomorphism ζ_3

$\zeta_3 : G_3^3 \to H,$

Homomorphism ζ_4

$\zeta_4 : G_1 \times G_4 \times G_1 \to H,$

Figure 5.47: Homomorphisms as defined in Input File

1.4.1 Access Structure

$$\Big((x_1, x_2) \wedge (x_1, x_4) \wedge (x_2, x_8, x_3)\Big) \vee \Big((x_5, x_6, x_7)\Big)$$

1.4.2 Constraints on Preimages

- $x_{1\zeta_2} = 1 \cdot x_{1\zeta_1}$
- $x_{2\zeta_4} = 1 \cdot x_{2\zeta_1}$

1.5 Relation

- $y_1 = \zeta_1(x_1, x_2)$
- $y_2 = \zeta_2(x_1, x_4)$
- $y_3 = \zeta_3(x_5, x_6, x_7)$
- $y_4 = \zeta_4(x_2, x_8, x_3)$

Figure 5.48: Access Structure and Constraint as defined in Input File

Now for the protocol, the compiler puts together the homomorphisms ζ_1 and ζ_2 into one single homomorphism. We call the homomorphisms which are the basic units for the protocol execution "atoms". An atom may be a composition of several homomorphisms from the user input or it may consist of a single homomorphism as specified in the user file. This decision is taken because of the constraints on the homomorphisms.

In our example the atoms for this protocol instance are the ones shown in figure 5.49.

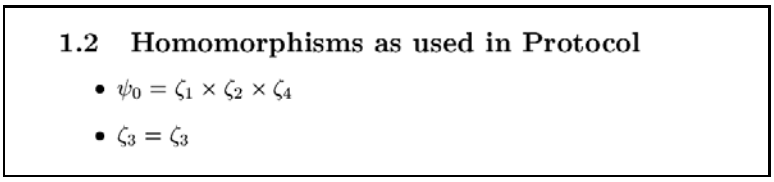


Figure 5.49: Homomorphisms as composed into atoms and used throughout the protocol

It might be confusing for the user that the homomorphisms used in the protocol specification are not the ones he has written in his input file. The only way to omit this problem would be to allow only constraints within a homomorphism or in other words to force the user to do this composition by himself that is now automated by the compiler. That would be no improvement as this possibility is included in the compiler as it is now as well.

5.7.3 Parsing concrete Homomorphisms and generating Code for these Functions

As described in section 3.2.3 it is possible to specify a homomorphism as a concrete function. Here is an example of such a specification:

```
DefineHomomorphism(zeta_2,
                   (x_[2..5]) |-> (h_1^(4*x_2) * h_2^(x_3*5 + x_4), h_3^x_5));
```

This function is used at several point in generating the target code depending on the target language.

If the target is LaTeX code it is used for the declaration part, where the same formula is needed as in the input file, but nevertheless it has to be formatted in the LaTeX way in order to prevent for example double superscript. In the protocol specification, the formula should appear each time the homomorphic function is calculated. In all these cases, the preimages in the formula defined in the input file should be replaced by the actual arguments.[15] Generating java the formula has to be used for generating the concrete implementation classes of the homomorphisms.

In order to make these different uses possible I decided to store the concrete homomorphism in the most general way: As an expression tree that represents the function. As the homomorphism function could map to a co-domain that consists of several groups, the concrete homomorphism is mapped into an expression tree for each element in the tuple. The parser builds up the tree and stores it in the symbol table in the corresponding homomorphism symbol.

Figure 5.50 shows the trees that are generated for this concrete homomorphic function.

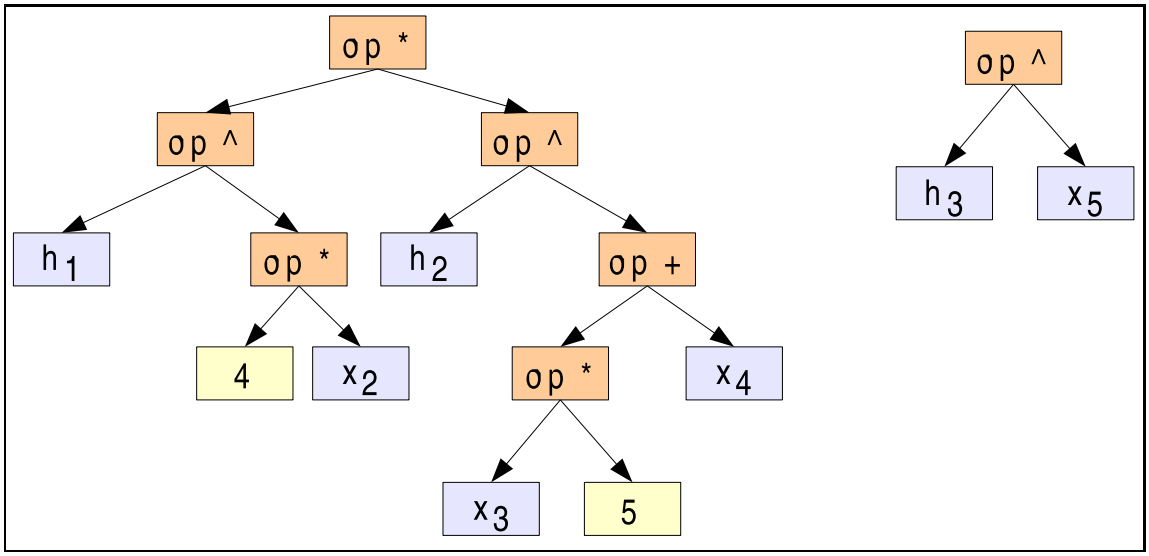


Figure 5.50: Expression Trees generated for the concrete Homomorphism

[15] This is the reason why the preimages have to be specified explitely in the definition instead of using the assignment e.g. $y_1^5 = h_1^{h_2 \cdot h_3}$. There would be no way to figure out, which are the preimages that should be replaced by the arguments and which are just constant group elements.

It is obvious that, in order to keep it as general as possible, we need a complete compiler within our compiler. The generation of the LaTeX string for this concrete homomorphism or the generation of the concrete homomorphism implementation class in java follows exactly the control flow as it was introduced in section 5.1. After the parsing process, the expression tree is checked by a small semantic checker whether it is a valid tree according to the group operations.[16] After the traversal of the semantic completion and analyze, a code generator which is another instance of the visitor pattern will generate the code.

This small compiler of its own can be found in the package `arithmeticExpressions`.

[16] An example of an invalid expression would be the following: $x, h \in G$, G is an group with group operation sign $+$, $x \mapsto (h * x)$. As h and x both are declared as group elements, the repeated operation with two group elements is not defined.

5.7.4 Formatting Variable Names for LaTeX Source Code

It is important for the user to be able to choose the names of the input elements such as homomorphisms, groups, group elements but maybe also of the temporary variables that are used during the protocol and the additional primitives used such as the secret sharing scheme, the commitment scheme and so on.

In the java context there are no problems with the variable names as long as they follow some basic rules[17] The LaTeX environment is not that liberal. The name `x_1_2` will in math mode be interpreted as a double subscript which is not permitted. Therefore the variable formatting for latex has to be considered.

The entity taking care of this problem is a static class called `LatexVariableFormatter` and is located in the package `util`. This class handles the cases with variables with two-dimensional indices, variable names that should be displayed in a special way such as greek letters and so on. For further maintenance it is important to know that the first part of a variable name goes from the beginning up to the first occurence of the underscore.[18]

This first part is reformatted according to the following rule: In the constant file for the whole project called `ZKCompiler_Constants` which is in the package `core` are two string arrays `MATH_SYMBOLS` and `MATH_SYMBOLS_CONVERTED` where the formatting rules are defined. In order to define further conventions one just has to append the string that has to be matched in the first array and the desired result in the second. Here is a short section from the definition of these two arrays:

```
String[] MATH_SYMBOLS = {
    "cPlus",
    "Z",
    "gamma" };
String[] MATH_SYMBOLS_CONVERTED = {
    "c^{+}",
    "\\mathbb{Z}",
    "\\gamma"};
```

[17] An identifier in Java has to start either with a letter or with a underscore. For the rest of the name letters, numbers and underscores are allowed in arbitrary order.

[18] It is possible to change this symbol that divides the variables into the name and the index part simply by changing the constant DIVISOR in the interface ZKCompiler_Constants where all constants are defined.

5.8 Overall Control Flow

After having introduced each of the main parts, I will end up this section about the design and implementation with an overview of the control flow.

Figure 5.51 shows the control flow from a high level point of view.

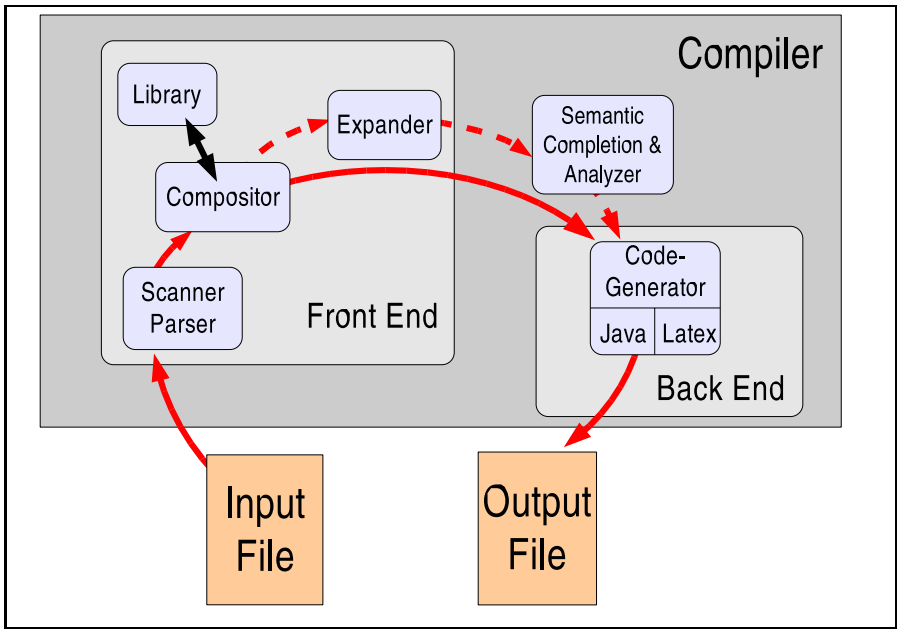


Figure 5.51: Control Flow from a high level Point of View

Here the two main parts - front end and back end - are shown as the grey boxes together with the most important classes shown in the bright blue ones. The control flow starts at the input file and passes in clockwise direction starting with the scanner and the parser at the beginning of the front end. Then the compositor is the next element that is involved. The compositor gets the protocol structure from the library where it is hard coded and puts together the different rounds of the protocol. It completes the symboltable and the intermediate representation which are now passed on.

From the compositor to the right there are two possible ways. If the user has chosen the compact mode and latex as target, it goes on directly with the code generation in the back end. In all other cases the intermediate representation is expanded and the semantic checks for type safety are performed by the Expander and the SemanticChecker.

The back end provides once more a choice in the control flow whether the intermediate representation and the symboltable are passed over to the java or to the latex code generator. These code generators directly write the chosen protocol instance into the output file.

Now lets have a closer look at the control flow. In figure 5.52 most entities that were described in the section about the design and implementation appear. The control flow starts as in the highlevel view with the input file in the lower left corner of figure 5.52. An additional element that was never mentionend up to now is right in the centre. It is the class `ZKCompiler` that contains the main method where everything starts and this is the class that hands over the arguments from one step in control flow to the other. The elements are again in clockwise order.

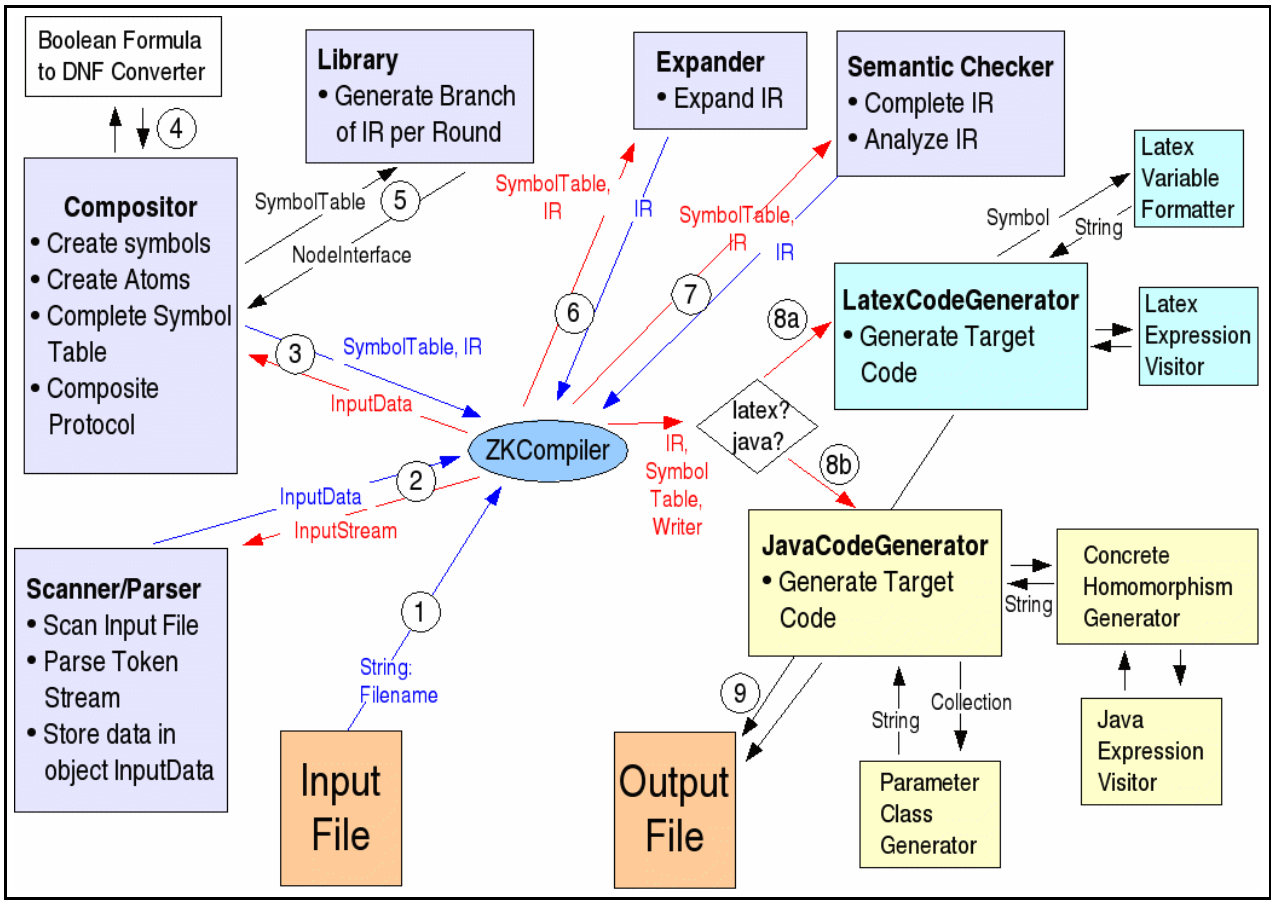


Figure 5.52: Control Flow in a more detailed view

1. The main method in the class `ZKCompiler` takes as argument the name of the file where the protocol instance is specified in the input language.

2. The class `ZKCompiler` opens an input stream, creates an instance of the ScannerParser class and initiates the parsing process. The scanner creates the stream of tokens that is consumed by the parser which stores the information into a class called `InputData`. Most data is stored simply in form of strings with the variable names. At the end of the parsing process, the `ZKCompiler` gets back all information from the input file stored in `InputData`.

3. The `InputData` goes on to the compositor. The compositor generates the symbols of the algebraic elements such as homomorphisms etc. and it starts to fill in all these elements in

the symbol table. The symbol table in contrast to the InputData stores all kind of symbols whereas the InputData is simply a sofisticated container for the names. The symbol table is now completed with all kind of symbols that are generated by the compositor which are needed for the protocol instance, but that were not specified in the input file. For example if several atoms[19] are defined, the compositor will add a symbol for each share that will be needed for this protocol.

4. If the access structure is specified in the relation section of the input file through a boolean formula, the compositor will use the static methods from the `BooleanFormulaToDNFConverter` as described in section 5.7.1 in order to translate it into the disjunctive normal form.

5. The compositor will then composite the intermediate representation (IR) depending on which protocol type is chosen. The IR is composited in the compact form where the structure of the homomorphisms and the groups is not yet taken into account. For each round of the protocol it will call the corresponding method in the library that returns a branch of the tree according to the protocol type and the round. These branches are put together by the compositor.

6. If the target is java or the protocol mode is verbose, the IR has to be expanded. Now the structure gets the precise form that fits together with the number of atoms and the structure of each atom and the groups involved in domain and co-domain. This step is left out in case where the target is latex and the mode is compact.

7. This step is taken only in the same cases as for item 6. The `SemanticChecker` completes the IR and analyzes it for correctness regarding the type safety.

8. Now there are two possible ways how this will continue depending on the output target.

   (a) If LaTeX code should be generated, the `LatexCodeGenerator` writes the declaration section into the filewriter that is passed as an argument. Each time a variable name has to be written, it will take its way through the class `LatexVariableFormatter`.

   If the user has defined some of the homomorphisms as concrete function, the latex expression for these homomorphic functions will be generated using the `LatexExpressionVisitor`.

   For the protocol execution itself the `LatexCodeGenerator` generates an instance of the NodeVisitor and gives the root of the IR as argument. The NodeVisitor will now iterate over the IR and write the code according to the respective node into the filewriter.

   (b) If the result should be an executable java program, a java code generator is instantiated. In case of concrete homomorphism it uses the `ConcreteHomomorphismGenerator` which will generate the concrete homomorphism implementation classes with the aid of the `JavaExpressionVisitor` that traverses the arithmetic expression. In order to generate the parameter classes for each send operation, the `ParameterClassGenerator` is involved in this process too. As the java back end has to generate not only one output file but one for the prover and one for the verifier it takes as input two filewriters from the `ZKCompiler`. It generates the code for these two files by instantiating the NodeVisitor for the java back end who iterates over the whole IR.

[19] About the connection between homomorphisms and atoms see section 5.7.2.

9. As a result we get one single output file in case of latex code or a folder with the java classes in it in case of java as target.

6 Java Interface

The java back end of the compiler produces runnable java source code that is based on an interface hierarchiy that I will explain in this section. The abstraction to work on interfaces instead of concrete implementation is very important for this compiler. The compiler takes as input different levels of abstraction and generates a protocol output of the according level. If a concrete homomorphism is specified, a concrete homomorphism implementation will be generated. But for the general case where a homomorphism is described only as a function that maps from domain groups to co-domain groups, it is only the protocol structure that is fix. The concrete implementations of the groups and homomorphisms can be exchanged or even implemented after compile time. The only requirements to these implementations is that they must implement resp. extend the abstraction defined in this diploma thesis.

6.1 Interface Hierarchy

First I want to introduce the three main elements — the homomorphic function, the group hierarchy and the group element hierarchy. These three elements are mandatory for all kind of zero-knowledge proof-of-knowledge protocols that can be generated using this compiler.

The homomorphism definition consists of an interface and an abstract class. Figure 6.53 shows the methods of these two types.

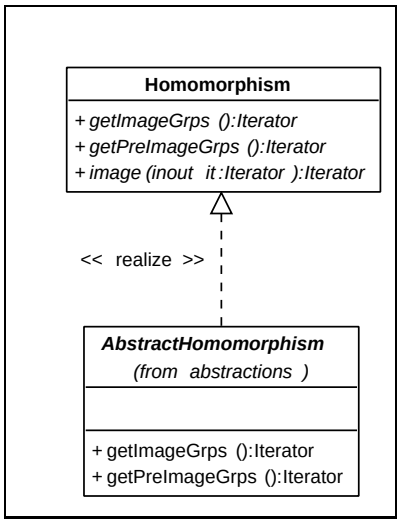


Figure 6.53: The Homomorphism Interface and the abstract class implementing the interface

Any homomorphism implementation used as constructor argument for the generated java files has to be an extension of the `AbstractHomomorphism` and therefore has to implement the homomorphism interface. All there has to implemented is some kind of constructor where the preimage and image groups are defined and the only method that has to be implemented is the `image` method that does the mapping of the elements from the domain into the elements of the co-domain.

The design of the group interface hierarchy is shown in figure 6.54. The parent interface for all extensions is the group interface itself that provides the basic group operations to perform the group operation, to get the neutral element and the inverse (if one does exist). For convenience reasons an additional method that performs the repeated group operation is provided.

This interface is extended by the interfaces of finite and infinite groups. Depending on this extension a further method is available to choose an element at random, either from the whole group or only from a certain interval. AdditiveGroup is another extension that is necessary for each generated protocol at least for the challenge and the shares. This interface is once more extended into the finite and the infinite part.

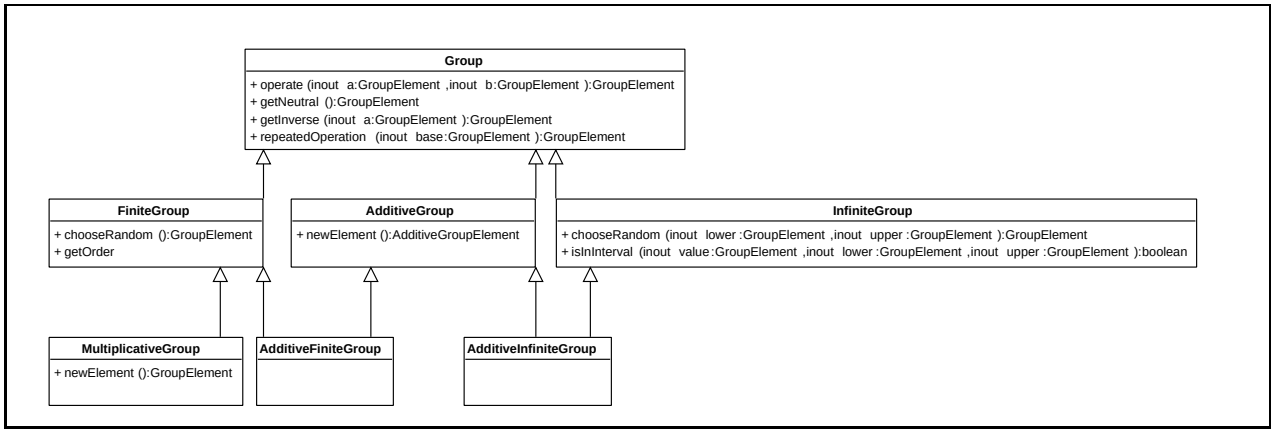


Figure 6.54: The Group Interface Hierarchy

The interface hierarchy for the group elements shown in figure 6.55 is very simple. There is the root interface group element whose only method is to get the corresponding group. For the two extensions there is the possibility to get a concrete value of type `Int` that was implemented for the idemix project by Dieter Sommer and belongs to the package `com.ibm.zurich.math.intmath`.

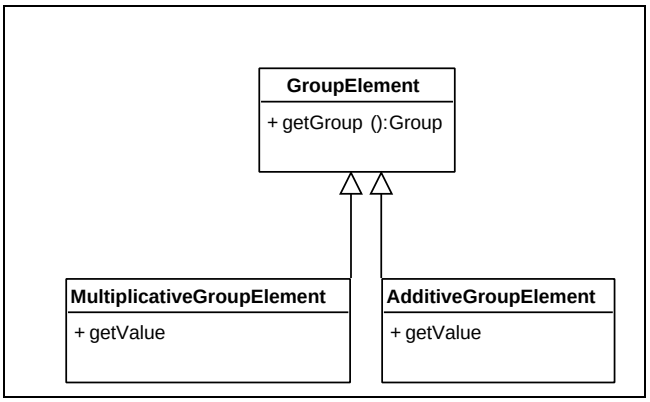


Figure 6.55: The Group Element Interface and its two Extensions

Figure 6.56 shows the interfaces for the commitment scheme that is used for the 2Σ protocol . The interface `Commitment` provides the two methods either to generate the randomness and on the other hand to commit to a certain value. In order to check whether the commitment was revealed correctly the method is invoked again with the value that has to be checked and the results of the two commit operations are compared using the `equals` method that is inherited from `Object` itself.

The other three interfaces `CommitedValue`, `Value` and `Randomness` are only marker interfaces.

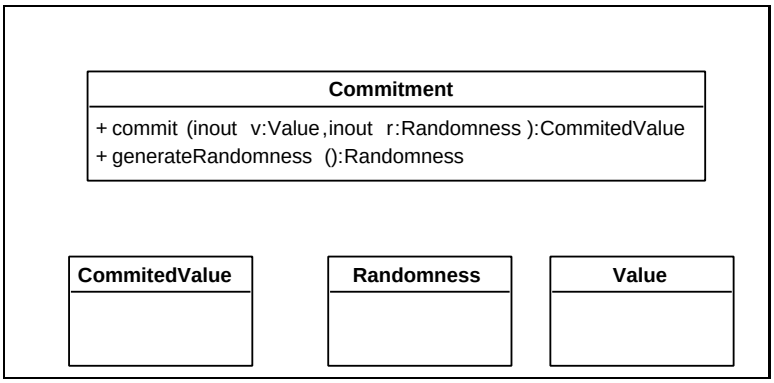


Figure 6.56: The Interfaces needed for the Commitment Primitive

The secret sharing scheme interface demands for the two methods `complete` that generates the shares from the challenge and the set of shares that was already chosen depending on the access structure, and the method `isConsistent` that checks a set of shares together with the challenge for consistency. These interfaces are shown in figure 6.57.

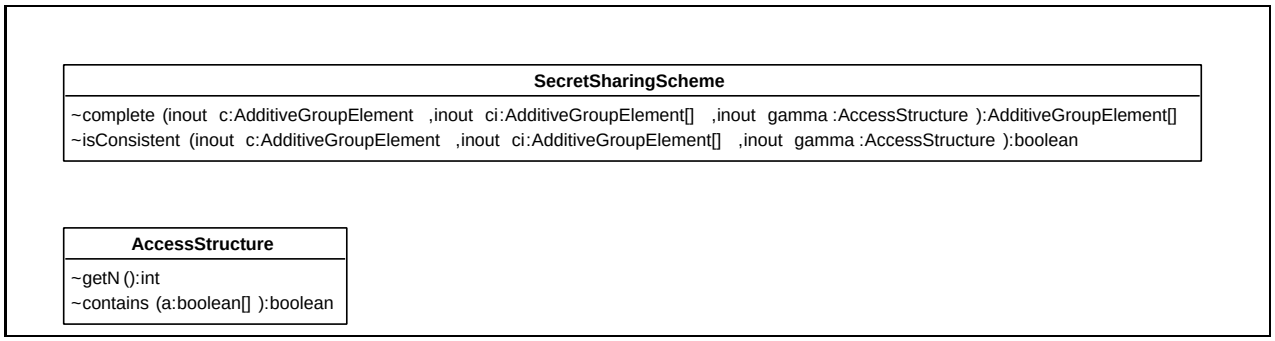


Figure 6.57: The Interfaces to modes the Secret Sharing Scheme

For the 2Σ protocol an instance of the strong rsa problem is needed. This is modelled as a factory object called `StrongRSAGenerator` that creates an instance of this strong rsa problem. The methods performed on this model are as shown in figure 6.58

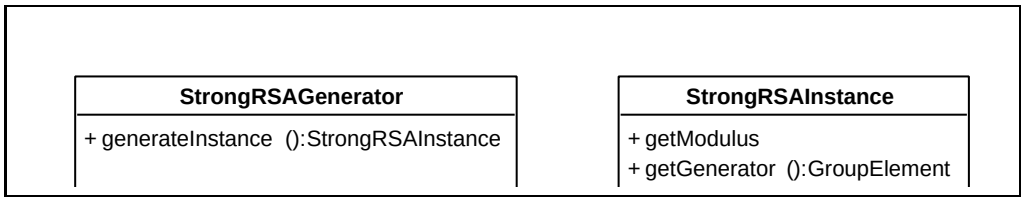


Figure 6.58: The Strong RSA Problem and its Generator modelled as Interfaces

6.2 Implementations

Up to now I was talking only about the interface level as the generated java source code operates only on interfaces. That is not completely true. There are several cases when the compiler works with concrete implementations too because of two reasons.

The first is that the protocol structure itself needs implementations independent of the instance which is chosen. This is the case for the additive groups that are always used for the challenge or if preimages are included in the index set and therefore an instance of the 2Σ protocol is generated. The 2Σ protocol works with additional homomorphisms in parallel with the user defined homomorphisms. These additional homomorphism are always of the same basic structure and are generated by the protocol itself. Therefore an implementation of this additional homomorphisms is provided. The constructor takes as input an array of group elements — the bases for the exponentiation. Of course it has to implement the `image` method from the interface `Homomorphism`.

The second reason is that in case where a user specifies for example the access structure, this is no longer an input parameter that will be given at runtime but it should be created already at compile time. Therefore an implementation is provided that only has to be instantiated. The same holds for the homomorphism in case where the user specifies it as a concrete function and the compiler therefore generates an implementation of this homomorphism.

Furthermore there are prototype implementations of some groups and primitives that were used for reasons of testing.

7 Conclusions and Future Work

In this section I summarize which of the goals of this project were attained, which new aims have been added during the development of the compiler, and which aspects are still left open. I want to arrive at a conclusion for which purposes this compiler should and can be used and which ones still have do be done by hand. Finally I will briefly describe my personal experiences during this diploma project.

The main goal of this diploma thesis according to the project description as given in appendix section A was "to develop a language for the specification [...] and a corresponding compiler which translates the protocol specifications into java source code [...]". This goal has been achieved completely. The compiler that was developed is able to generate not even java source code but also a protocol specification in LaTeX , which during the project turned out to be at least as important as the java output. The LaTeX code generated provides, besides the description of the various rounds of a protocol instance, a declaration section that summarizes the protocol specification and some conclusions from this specification in a well-structured form for further evaluation.

The specification language develped in collaboration with my supervisors is able to express the features that are used for these types of zero-knowledge proof-of-knowledge protocols. The input files written in the specification language are easy to understand thanks to a comprehensible language style that is strongly related to other programming and specification languages.

The architecture of the compiler follows an modular object-oriented design and makes extensive use of well-known design patterns. It is strongly hierarchically structured and packaged to facilitate maintenance. The interface design used by the java back end has been carefully designed and takes advantage of the abstraction mechanisms provided by java.

The compiler in the actual state is a useful tool for generating and specifying new protocol instances that provide different features, such as interval checks or preimage constraints, based on either the Σ or the 2Σ protocol working with arbitrary homomorphisms, which can be specified simply by the mapping from domain groups to co-domain groups or by using a concrete homomorphic function. It saves a lot of work and eliminates the risk of making mistakes during the generation of the specification. The java source code generated can be used for performance testing and provides the entire range of flexibility for using different kinds of homomorphisms and group implementations.

For lack of time, the aim of making further statements about the protocol instances generated e.g. whether they fulfill the soundness property or other semantic properties, has been dropped. In order to support decisions on these semantic properties a section in the protocol description that shows all properties of the protocol instance in one single formula if the proof is accepted has been added in LaTeX output mode.

The compiler provides several access points for future work. The decision problem on the semantic properties could be implemented, based on knowledge of the protocol instance collected in the symbol table by deducing further statements from the collision equation. Further optimization steps could be added to improve the efficiency of the java source code generated.

In order to use this compiler for further prototyping, additional implementations of instances of secret-sharing schemes, commitment schemes, different types of groups and so on would be necessary so that a rich toolbox is available for testing.

If at some point in time a decision is made that these protocols should run in other target environments, such as embedded systems, additional back ends would possibly be needed that generate different code. Such back ends would fit directly into the compiler design.

The four months I was working on my diploma thesis were a very interesting and challenging time for me. The strong connection between understanding theoretical ideas of the cryptographic background and their translation into terms of software engineering were a real highlight. I have learned a lot about zero-knowledge proof-of-knowledge protocols, but also about the process of developing software and about language and compiler design.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers, Principles, Techniques, and Tools*, Addison-Wesley, 1996

[2] Ronald Cramer, Ivan Damgård, Berry Schoenmaker, *Proofs of Partial Knowledge and Simplified Design of Witness Hiding Protocols*, CRYPTO '94, Vol. 839 of Lecture Notes in Computer Science, Springer-Verlag, 1994, pp. 174 - 187

[3] Ivan Damgård, *On* Σ *- protocols*, CPT 2002

[4] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995

[5] Louis C.Guillou, Jean-Jacques Quisquater, *A practical zero-knowledge protocol fitted to security microprocessor minimizing both transmission and memory*, Advances in Cryptology - EUROCRYPT '88, p.123-128, 1988

[6] Bertrand Meyer, *Object-oriented software construction*, Prentice Hall, 1997

[7] Claus-Peter Schnorr, *Efficient signature generation for smart cards*, Journal of Cryptology, 4: p.161-174, 1991.

[8] Niklaus Wirth, *Grundlagen und Techniken des Compilerbaus*, Addison-Wesley, 1996

A Project Description

Diploma Work: Development of a Compiler for Zero-Knowledge Proof-of-Knowledge Protocols

The goal of this diploma work is to develop a language for the specification of so called "zero-knowledge proofs of knowledge of a preimage of a homomorphism" protocols and a corresponding compiler which translates protocol specifications into JAVA source code which realizes the protocol.

"Zero-knowledge proofs of knowledge of a preimage of a homomorphism" are protocols between two parties called prover and verifier. Informally, these two-party protocols have the following properties: a function $f()$ (group homomorphism) and a y are given to both the prover and verifier. Additionally the prover is given a secret value x such that $y = f(x)$. At the end of a successful protocol execution it holds that

- the verifier can be assured, that the prover indeed knows the secret corresponding to y with respect to f,
- the prover can be assured, that the verifier did not learn anything about the secret value x.

Several basic protocols with these properties do exist. Furthermore, by combining such basic protocols knowledge of relations among the secrets can be proved. For instance, the prover can prove that, given y_1, y_2, $f_1()$, $f_2()$, and two integers a and b, it knows secret values x_1 and x_2 such that the relation $ax_1 + bx_2 = 0$ is fulfilled (whereas the verifier does not gain knowledge about the values of x_1 and x_2). Such protocols are fascinating from a purely theoretical point of view and considerable theoretical research was conducted on the topic. From a practical point of view these protocols play an important role as sub-protocols in higher level protocols, such as novel anonymous public key infrastructures and secure multi-party computations.

Currently such systems and protocols are being developed at the IBM Zurich Lab (for an overview please refer to http://www.zurich.ibm.com/security/idemix). Hence the aim arises not only to consider these protocols from a theoretical point of view but also to actually implement software systems that make use of zero-knowledge proofs of knowledge. As a matter of fact, for someone having certain cryptographic knowledge, given some function $g()$ it is fairly easy to choose which protocol of many possible ones is appropriate to prove knowledge of secret with respect to $g()$. However, it is quite *time consuming* and *error prone* to actually generate a description (implementation) of the chosen protocol for instance in the JAVA programming language. This observation constitutes the motivation of the diploma work, which has the objective to automate the translation process from a protocol specification to the protocol description / implementation. More precisely, the goal of the diploma-work is to

- create in collaboration with a cryptographer a *protocol specification language* for zero-knowledge proofs of knowledge of a preimage of a homomorphism..
- design and implement a *compiler* which translates protocol specifications into a target language, e.g., JAVA.

We are looking for students with theoretical and practical knowledge in compiler design / implementation and good JAVA programming skills. While no cryptographic knowledge is required, the student should be interested in learning some of the cryptographic theory underlying the diploma work.

For more information please contact: IBM Zurich Research Lab, Endre Bangerter, Saeumerstr. 4, 8803 Rueschlikon, Switzerland. e-mail: eba@zurich.ibm.com

B Example of a protocol specification as used for idemix system

$$PK\Big\{(\alpha,\beta_1,\ldots,\beta_7,\gamma,\delta,\varepsilon,\eta,\zeta,\xi,\varphi,\kappa_1,\ldots,\kappa_3,\psi_1,\ldots,\psi_3,\mu_1,\ldots,\mu_4,\nu_0,\ldots,\nu_4,\vartheta_0,\ldots,\vartheta_3\rho,\chi):$$

$$d_{O_j}^2 = (A^2)^{\varepsilon}(\frac{1}{a_{O_j}^2})^{\alpha}(\frac{1}{w_{O_j}^2})^{\beta_5}(\frac{1}{\nu_{(1,O_j)}^{2a_1}})^{\beta_1}(\frac{1}{\nu_{(2,O_j)}^{2a_1}})^{\beta_2}(\frac{1}{\nu_{(3,O_j)}^{2a_1}})^{\beta_3}(\frac{1}{\tilde{\nu}_{O_j}^{2a_2}})^{\beta_4}(\frac{1}{\widetilde{w}_{O_j}^{2a_3}})^{\beta_6}(\frac{1}{z_{O_j}^{2a_4}})^{\beta_7}(\frac{1}{b_{O_j}^2})^{\gamma}(\frac{1}{h_{O_j}^2})^{\delta}\wedge$$

$$B^2 = (h_{O_j}^2)^{\eta}(g_{O_j}^2)^{\zeta}\ \wedge\ 1 = (B^2)^{\varepsilon}(\frac{1}{h_{O_j}^2})^{\delta}(\frac{1}{g_{O_j}^2})^{\xi}\wedge$$

$$\Big[v_{O_j} = \mathfrak{C}_u^{\varepsilon}(\frac{1}{h_{O_j}})^{\vartheta_0}\ \wedge\ \mathfrak{C}_r = h_{O_j}^{\vartheta_1}g_{O_j}^{\vartheta_2}\ \wedge\ 1 = \mathfrak{C}_r^{\varepsilon}(\frac{1}{h_{O_j}})^{\vartheta_3}(\frac{1}{g_{O_j}})^{\vartheta_0}\wedge\Big]_{b_0}$$

$$[M = g^{\beta_1}\ \wedge\ g^r = (g^c)^{\beta_2}g^{\beta_3}\wedge]_{b_1}$$

$$[L = g^{\beta_4}\wedge]_{b_2}$$

$$\big[R_0^2(g_{O_v}^2)^{edate} = (g_{O_v}^2)^{\beta_6}(h_{O_v}^2)^{\nu_0}\ \wedge\ R_1^2 = (g_{O_v}^2)^{\mu_1}(h_{O_v}^2)^{\nu_1}\wedge$$

$$R_2^2 = (g_{O_v}^2)^{\mu_2}(h_{O_v}^2)^{\nu_2}\ \wedge\ R_3^2 = (g_{O_v}^2)^{\mu_3}(h_{O_v}^2)^{\nu_3}\ \wedge\ R_4^2 = (g_{O_v}^2)^{\mu_4}(h_{O_v}^2)^{\nu_4}\wedge$$

$$(R_0^2) = (R_1^2)^{\mu_1}(R_2^2)^{\mu_2}(R_3^2)^{\mu_3}(R_4^2)^{\mu_4}(h_{O_v}^2)^{\rho}\wedge\big]_{b_3}$$

$$[g^{stype} = g^{\beta_7}\wedge]_{b_4}$$

$$[K_{(U,O_v)} = g^{\alpha}h^{\varphi}\wedge]_{b_5}$$

$$[K_{(U,R)} = g^{\alpha}h^{\beta_5}h_1^{\psi_1}\ =\ \wedge\widetilde{W}_1^2 = (g_R^2)^{\psi_2}\wedge\widetilde{W}_2^2 = (h_R^2)^{\psi_2}((1+n_R)^2)^{\psi_1}\wedge$$

$$\widetilde{W}_3^2 = ((y_R z_R^{\mathcal{H}(\widetilde{W}_1,\widetilde{W}_2,m_R)})^2)^{\psi_2}\wedge C_R^2 = (g_{O_v}^2)^{\psi_1}(h_{O_v}^2)^{\psi_3}\wedge\psi_1\in[-n_R/2,n_R/2]\wedge]_{b_6}$$

$$[K_{(U,T)} = g^{\alpha}h_1^{\kappa_1}\ \wedge\ W_1^2 = (g_T^2)^{\kappa_2}\ \wedge\ W_2^2 = (h_T^2)^{\kappa_2}((1+n_T)^2)^{\kappa_1}\wedge$$

$$W_3^2 = ((y_T z_T^{\mathcal{H}(W_1,W_2,m_T)})^2)^{\kappa_2}\wedge C_T^2 = (g_{O_v}^2)^{\kappa_1}(h_{O_v}^2)^{\kappa_3}\wedge\kappa_1\in[-n_T/2,n_T/2]\wedge]_{b_7}$$

$$\alpha,\beta_1,\ldots,\beta_7\in\Gamma\ \wedge\ \gamma\in\Delta\ \wedge\ \varepsilon\in\Lambda$$

$$\Big\}\ .$$

C Examples of Input Files and Generated Files

C.1 Instance of the Σ protocol

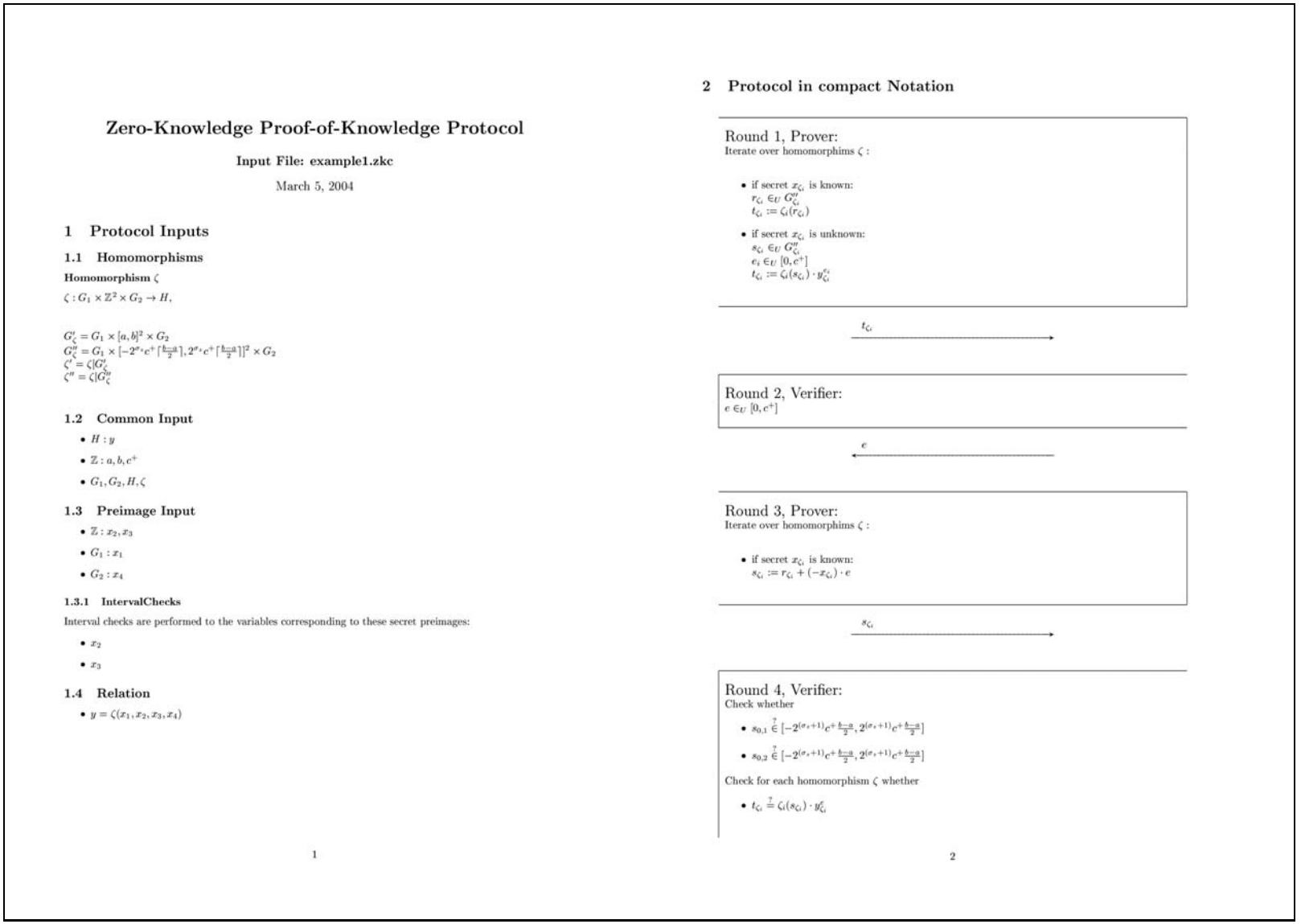
Example 1 shows the specification of an instance of the Σ protocol . The generated LaTeX output is shown in figure C.59.

```
// Declarations
Group (G_1,+), (G_2,+);
Group (H,*);
GroupElement x_[1..4];
GroupElement y;
Homomorphism zeta;
IntegerInterval i_1, i_2;
IntegerConstant a, b, c, d;

// Assignments
AssignGroupMember(G_1, x_1);
AssignGroupMember(Z, {x_2, x_3});
AssignGroupMember(G_2, x_4);
AssignGroupMember(H, y);

// Definitions
DefineHomomorphism(zeta, G_1 # (i_1 subset Z) # (i_1 subset Z) # G_2 -> H);
DefineIntegerInterval(i_1, (a,b)), (i_2, (c,d));

// Protocol Specification
SpecifyProtocol [
    Relation [
        CommonInput = {(zeta, (y))};
        PreimageInput = {(x_1, x_2, x_3, x_4)};
    ]

    IntervalCheck = {x_2, x_3};

    ParameterNames {
        VariableC = e;
    }

    Target = LATEX;
    Layout = COMPACT;
]
```

Here are some of the remarks on this example:

- The protocol instance works with the additive infinite group $\mathbb{Z}$ that does not have to be declared as it is always already instantiated.

- The homomorphism is specified as a mapping of groups what we called the abstract structure. Therefore intervals for the infinite groups can be specified directly in the definition.
- For the specification of the relation the enumerative description is chosen that splits up the data into the common and the preimage input.
- To the symbols `x_2` and `x_3` that belong to the infinite group an additional interval check will be performed.
- As a integer constant with the name c is declared and used, the name for the challenges in the protocol has to be changed. Therefore the parameter name `VariableC` is set to another letter.



Zero-Knowledge Proof-of-Knowledge Protocol

Input File: example1.zkc

March 5, 2004

1 Protocol Inputs

1.1 Homomorphisms

Homomorphism ζ

$\zeta : G_1 \times \mathbb{Z}^2 \times G_2 \to H,$

$G'_\zeta = G_1 \times [a,b]^2 \times G_2$
$G''_\zeta = G_1 \times [-2^{\sigma_\epsilon} c^+ \lceil \frac{b-a}{2} \rceil, 2^{\sigma_\epsilon} c^+ \lceil \frac{b-a}{2} \rceil]^2 \times G_2$
$\zeta' = \zeta | G'_\zeta$
$\zeta'' = \zeta | G''_\zeta$

1.2 Common Input

- $H : y$
- $\mathbb{Z} : a, b, c^+$
- G_1, G_2, H, ζ

1.3 Preimage Input

- $\mathbb{Z} : x_2, x_3$
- $G_1 : x_1$
- $G_2 : x_4$

1.3.1 IntervalChecks

Interval checks are performed to the variables corresponding to these secret preimages:

- x_2
- x_3

1.4 Relation

- $y = \zeta(x_1, x_2, x_3, x_4)$

2 Protocol in compact Notation

Round 1, Prover:
Iterate over homomorphims ζ :

- if secret x_{ζ_i} is known:
  $r_{\zeta_i} \in_U G''_{\zeta_i}$
  $t_{\zeta_i} := \zeta_i(r_{\zeta_i})$
- if secret x_{ζ_i} is unknown:
  $s_{\zeta_i} \in_U G''_{\zeta_i}$
  $e_i \in_U [0, c^+]$
  $t_{\zeta_i} := \zeta_i(s_{\zeta_i}) \cdot y_{\zeta_i}^{e_i}$

t_{ζ_i} →

Round 2, Verifier:
$e \in_U [0, c^+]$

← e

Round 3, Prover:
Iterate over homomorphims ζ :

- if secret x_{ζ_i} is known:
  $s_{\zeta_i} := r_{\zeta_i} + (-x_{\zeta_i}) \cdot e$

s_{ζ_i} →

Round 4, Verifier:
Check whether

- $s_{0,1} \stackrel{?}{\in} [-2^{(\sigma_\epsilon+1)} c^+ \frac{b-a}{2}, 2^{(\sigma_\epsilon+1)} c^+ \frac{b-a}{2}]$
- $s_{0,2} \stackrel{?}{\in} [-2^{(\sigma_\epsilon+1)} c^+ \frac{b-a}{2}, 2^{(\sigma_\epsilon+1)} c^+ \frac{b-a}{2}]$

Check for each homomorphism ζ whether

- $t_{\zeta_i} \stackrel{?}{=} \zeta_i(s_{\zeta_i}) \cdot y_{\zeta_i}^e$

Figure C.59: Generated LaTeX code from Example Input File 1

C.2 Instance of the 2Σ protocol

In Example 2 that is shown in figure C.60 an instance of the 2Σ protocol is specified. Some remarks to this example:

```
// Declarations
Group (G,+), Zm;
Group (H_[1..2],*);
GroupElement x_[1..8];
GroupElement y_[1..4];
Homomorphism zeta_[1..3];
IntegerInterval i;
IntegerConstant a, b, e, f;


// Assignments
AssignGroupMember(G, {x_[5..6]});
AssignGroupMember(Z, x_3);
AssignGroupMember(Zm, {x_[1..2], x_4, x_[7..8]});
AssignGroupMember(H_1, {y_1, y_[3..4]});
AssignGroupMember(H_2, y_2);

// Definitions
DefineHomomorphism(zeta_1, Zm # Zm  # (i subset Z) # Zm # G -> H_1 # H_2);
DefineHomomorphism(zeta_2, G # Zm -> H_1);
DefineHomomorphism(zeta_3, Zm -> H_1);
DefineIntegerInterval(i, (a,b));

// Protocol Specification
SpecifyProtocol [
    Relation  = [(y_1^ e, y_2^(3*f)) = zeta_1(x_1, x_2, x_3, x_4, x_5)]
                && ([(y_3) = zeta_2(x_6, x_7)] || [(y_4) = zeta_3(x_8)]);

    IndexSet = {x_1, x_2, x_3, x_7};

    IntervalCheck = {x_3};

    ParameterNames {
        VariableR = u;
    }

    Target = LATEX;
]
```

- In this example we use an additive finite group with the order m. The name of the group has

therefore to be `Zm`. This group needs no specification of the group operation as this type of group names is predefined as an additive group.

- We now have several homomorphisms and define the relation as a boolean formula with these three homomorphisms as literals.
- Variables that belong to `Zm` and `Z` are included into the index set. Therefore an instance of the 2Σ protocol will be generated.
- In the relation section, the image part of the homomorphism may be not only a single group member but can also be an expression as long as it is consistent with the groups that were defined for the co-domain.

Zero-Knowledge Proof-of-Knowledge Protocol

Input File: example2.zkc

March 5, 2004

1 Protocol Inputs

1.1 Homomorphisms

Homomorphism ζ_1

$\zeta_1 : Z_m^2 \times Z \times Z_m \times G \to H_1 \times H_2,$

$G'_{\zeta_1} = Z_m^2 \times [a, b] \times Z_m \times G$
$G''_{\zeta_1} = Z_m^2 \times [-2^{\ell_e} c^+ \lceil \frac{k-1}{2} \rceil, 2^{\ell_e} c^+ \lceil \frac{k-1}{2} \rceil] \times Z_m \times G$
$\zeta'_1 = \zeta_1 |G'_{\zeta_1}$
$\zeta''_1 = \zeta_1 |G''_{\zeta_1}$

IndexSet: $[x_1, x_2, x_3]$

Homomorphism ζ_2

$\zeta_2 : G \times Z_m \to H_1,$

IndexSet: $[x_7]$

Homomorphism ζ_3

$\zeta_3 : Z_m \to H_1,$

1.2 Common Input

- $H_1 : y_1, y_3, y_4$
- $Z : a, b, c^+, c^-, f$
- $H_2 : y_2$
- Commitment Scheme, G, H_1, H_2, Secret Sharing Scheme $\Theta, Z_m, \zeta_1, \zeta_2, \zeta_3$

1.3 Preimage Input

- $Z : x_5$
- $Z_m : x_1, x_2, x_4, x_7, x_8$
- $G : x_5, x_6$
- Qualified Set A

1.3.1 Access Structure

$\Big((x_1, x_2, x_3, x_4, x_5) \wedge (x_6, x_7)\Big) \vee \Big((x_1, x_2, x_3, x_4, x_5) \wedge (x_8)\Big)$

1.3.2 IntervalChecks

Interval checks are performed to the variables corresponding to these secret preimages:

- x_5

1.4 Relation

- $(y_1{}^c, y_2{}^{(3\cdot f)}) = \zeta_1(x_1, x_2, x_3, x_4, x_5)$
- $y_3 = \zeta_2(x_6, x_7)$
- $y_4 = \zeta_3(x_8)$

2 Protocol Properties

$$\Big[\Big((y_1{}^c)^{\gamma_6}, (y_2{}^{(3\cdot f)})^{\gamma_6}\Big) = \zeta_1(x_1, x_2, x_3, x_4, x_5)\Big) \wedge \Big((y_3)^{\gamma_5} = \zeta_2(x_6, x_7)\Big)$$
$$\wedge \Big(\sigma_{0,2} \in [-2^{(\ell_\sigma+2)} c^+ \tfrac{k-1}{2}, 2^{(\ell_\sigma+2)} c^+ \tfrac{k-1}{2}]\Big) \wedge \Big(\gamma_6 | \sigma_{0,0}\Big) \wedge \Big(\gamma_6 | \sigma_{0,1}\Big) \wedge \Big(\gamma_1 | \sigma_{1,1}\Big)\Big]$$
$$\vee \Big[\Big((y_1{}^c)^{\gamma_6}, (y_2{}^{(3\cdot f)})^{\gamma_6}\Big) = \zeta_1(x_1, x_2, x_3, x_4, x_5)\Big) \wedge \Big((y_4)^{\gamma_5} = \zeta_3(x_8)\Big)$$
$$\wedge \Big(\sigma_{0,2} \in [-2^{(\ell_\sigma+2)} c^+ \tfrac{k-1}{2}, 2^{(\ell_\sigma+2)} c^+ \tfrac{k-1}{2}]\Big) \wedge \Big(\gamma_6 | \sigma_{0,0}\Big) \wedge \Big(\gamma_6 | \sigma_{0,1}\Big) \wedge \Big(\gamma_6 | \sigma_{0,2}\Big)\Big]$$

Figure C.60: Generated LaTeX code from Example Input File 2

3 Protocol in verbose Notation

Round 1, Verifier:
$(n, g) := I_{SRSA}(\sigma_*)$.
$\rho_0 \in_U [0, 2^{\sigma_*} \lfloor \frac{n}{4} \rfloor]$
$\rho_1 \in_U [0, 2^{\sigma_*} \lfloor \frac{n}{4} \rfloor]$
$\rho_2 \in_U [0, 2^{\sigma_*} \lfloor \frac{n}{4} \rfloor]$
$g_0 := g^{\rho_0}$
$g_1 := g^{\rho_1}$
$g_2 := g^{\rho_2}$

n, g_0, g_1, g_2, g ←

Round 2, Prover:

- if secret $(x_1, x_2, x_3, x_4, x_5)$ corresponding to ζ_1 is known:
  $u_{0,0} \in_U \mathbb{Z}_m$
  $u_{0,1} \in_U \mathbb{Z}_m$
  $u_{0,2} \in_U [-2^{\sigma_*} c^+ \frac{b-a}{2}, 2^{\sigma_*} c^+ \frac{b-a}{2}]$
  $u_{0,3} \in_U \mathbb{Z}_m$
  $u_{0,4} \in_U G$
  $(t_{0,0}, t_{0,1}) := \zeta_1(u_{0,0}, u_{0,1}, u_{0,2}, u_{0,3}, u_{0,4})$
  $\hat{x}_0 \in_U [0, \lfloor \frac{n}{4} \rfloor]$
  $\hat{y}_0 := \hat{\zeta}_1(x_1, x_2, x_3, \hat{x}_0)$
  $\hat{u}_0 \in_U [-2^{\sigma_*} c^+ \lfloor \frac{n}{4} \rfloor, 2^{\sigma_*} c^+ \lfloor \frac{n}{4} \rfloor]$
  $\hat{t}_0 := \hat{\zeta}_1(u_{0,0}, u_{0,1}, u_{0,2}, \hat{u}_0)$
  $r_{\hat{y}_0} :=$ commitRand()
  $r_{\hat{t}_0} :=$ commitRand()
  $\bar{y}_0 := \text{commit}(\hat{y}_0, r_{\hat{y}_0})$
  $\bar{t}_0 := \text{commit}(\hat{t}_0, r_{\hat{t}_0})$
- if secret $(x_1, x_2, x_3, x_4, x_5)$ corresponding to ζ_1 is unknown:
  $s_{0,0} \in_U \mathbb{Z}_m$
  $s_{0,1} \in_U \mathbb{Z}_m$
  $s_{0,2} \in_U [-2^{\sigma_*} c^+ \frac{b-a}{2}, 2^{\sigma_*} c^+ \frac{b-a}{2}]$
  $s_{0,3} \in_U \mathbb{Z}_m$
  $s_{0,4} \in_U G$
  $c_0 \in_U [0, c^+]$
  $(t_{0,0}, t_{0,1}) := \zeta_1(s_{0,0}, s_{0,1}, s_{0,2}, s_{0,3}, s_{0,4}) \cdot (y_1{}^{c c_0}, y_2{}^{(3 \cdot f)^{c_0}})$
  $\hat{x}_0 \in_U [0, \lfloor \frac{n}{4} \rfloor]$
  $\hat{y}_0 := g^{\hat{x}_0}$
  $\hat{s}_0 \in_U [-2^{\sigma_*} c^+ \lfloor \frac{n}{4} \rfloor, 2^{\sigma_*} c^+ \lfloor \frac{n}{4} \rfloor]$
  $\hat{t}_0 := \hat{\zeta}_1(s_{0,0}, s_{0,1}, s_{0,2}, \hat{s}_0) \cdot \bar{y}_0^{c c_0}$
  $r_{\hat{y}_0} :=$ commitRand()
  $r_{\hat{t}_0} :=$ commitRand()
  $\bar{y}_0 := \text{commit}(\hat{y}_0, r_{\hat{y}_0})$
  $\bar{t}_0 := \text{commit}(\hat{t}_0, r_{\hat{t}_0})$

- if secret (x_6, x_7) corresponding to ζ_2 is known:
  $u_{1,0} \in_U G$
  $u_{1,1} \in_U \mathbb{Z}_m$
  $t_{1,0} := \zeta_2(u_{1,0}, u_{1,1})$
  $\hat{x}_1 \in_U [0, \lfloor \frac{n}{4} \rfloor]$
  $\hat{y}_1 := \hat{\zeta}_2(x_7, \hat{x}_1)$
  $\hat{u}_1 \in_U [-2^{\sigma_*} c^+ \lfloor \frac{n}{4} \rfloor, 2^{\sigma_*} c^+ \lfloor \frac{n}{4} \rfloor]$
  $\hat{t}_1 := \hat{\zeta}_2(u_{1,1}, \hat{u}_1)$
  $r_{\hat{y}_1} :=$ commitRand()
  $r_{\hat{t}_1} :=$ commitRand()
  $\bar{y}_1 := \text{commit}(\hat{y}_1, r_{\hat{y}_1})$
  $\bar{t}_1 := \text{commit}(\hat{t}_1, r_{\hat{t}_1})$
- if secret (x_6, x_7) corresponding to ζ_2 is unknown:
  $s_{1,0} \in_U G$
  $s_{1,1} \in_U \mathbb{Z}_m$
  $c_1 \in_U [0, c^+]$
  $t_{1,0} := \zeta_2(s_{1,0}, s_{1,1}) \cdot y_3{}^{c_1}$
  $\hat{x}_1 \in_U [0, \lfloor \frac{n}{4} \rfloor]$
  $\hat{y}_1 := g^{\hat{x}_1}$
  $\hat{s}_1 \in_U [-2^{\sigma_*} c^+ \lfloor \frac{n}{4} \rfloor, 2^{\sigma_*} c^+ \lfloor \frac{n}{4} \rfloor]$
  $\hat{t}_1 := \hat{\zeta}_2(s_{1,1}, \hat{s}_1) \cdot \bar{y}_1^{c_1}$
  $r_{\hat{y}_1} :=$ commitRand()
  $r_{\hat{t}_1} :=$ commitRand()
  $\bar{y}_1 := \text{commit}(\hat{y}_1, r_{\hat{y}_1})$
  $\bar{t}_1 := \text{commit}(\hat{t}_1, r_{\hat{t}_1})$

- if secret x_8 corresponding to ζ_3 is known:
  $u_{2,0} \in_U \mathbb{Z}_m$
  $t_{2,0} := \zeta_3(u_{2,0})$
- if secret x_8 corresponding to ζ_3 is unknown:
  $s_{2,0} \in_U \mathbb{Z}_m$
  $c_2 \in_U [0, c^+]$
  $t_{2,0} := \zeta_3(s_{2,0}) \cdot y_4{}^{c_2}$

$t_{0,0}, t_{0,1}, t_{1,0}, t_{2,0}, \bar{t}_0, \bar{t}_1, \bar{y}_0, \bar{y}_1$ →

Round 3, Verifier:
$c \in_U [0, c^+]$

c ←

Round 4, Prover:

$(c_0, c_1, c_2) := \text{complete}(c, \{[c_i]_4\}, \Gamma^*(n))$

- if secret $(x_1, x_2, x_3, x_4, x_5)$ corresponding to ζ_1 is known:
  $(s_{0,0}, s_{0,1}, s_{0,2}, s_{0,3}, s_{0,4}) := (u_{0,0}, u_{0,1}, u_{0,2}, u_{0,3}, u_{0,4}) + ((-((x_1, x_2, x_3 + \frac{b+a}{2}, x_4, x_5)))) \cdot c_0$
  $\hat{s}_0 := \hat{u}_0 + (-(\hat{x}_0)) \cdot c_0$

- if secret (x_6, x_7) corresponding to ζ_2 is known:
  $(s_{1,0}, s_{1,1}) := (u_{1,0}, u_{1,1}) + ((-((x_6, x_7)))) \cdot c_1$
  $\hat{s}_1 := \hat{u}_1 + (-(\hat{x}_1)) \cdot c_1$

- if secret x_8 corresponding to ζ_3 is known:
  $s_{2,0} := u_{2,0} + ((-(x_8))) \cdot c_2$

$s_{0,0}, s_{0,1}, s_{0,2}, s_{0,3},$
$s_{0,4}, s_{1,0}, s_{1,1}, s_{2,0},$
$\hat{s}_0, \hat{s}_1, c_0, c_1,$
c_2

→

Round 5, Verifier:

←

ρ_0, ρ_1, ρ_2

Round 6, Prover:

Check whether

- $(g_0, g_1, g_2) \stackrel{?}{=} (g^{\rho_0}, g^{\rho_1}, g^{\rho_2})$

$\hat{t}_0, \hat{t}_1, r_{\hat{t}_0}, r_{\hat{t}_1},$
$\hat{y}_0, \hat{y}_1, r_{\hat{y}_0}, r_{\hat{y}_1}$

→

Round 7, Verifier:

Check whether

- $s_{0,2} \stackrel{?}{\in} [-2^{(\sigma_*+1)c^+ \frac{b-a}{2}}, 2^{(\sigma_*+1)c^+ \frac{b-a}{2}}]$

Check whether

- $(t_{0,0}, t_{0,1}) \stackrel{?}{=} \zeta_1(s_{0,0}, s_{0,1}, s_{0,2} + (\frac{b+a}{2}) \cdot c_0, s_{0,3}, s_{0,4}) \cdot (y_1{}^{e \cdot c_0}, y_2{}^{(3 \cdot f) c_0})$
- $t_{1,0} \stackrel{?}{=} \zeta_2(s_{1,0}, s_{1,1}) \cdot y_3{}^{c_1}$
- $t_{2,0} \stackrel{?}{=} \zeta_3(s_{2,0}) \cdot y_4{}^{c_2}$
- $\hat{t}_0 \stackrel{?}{=} \hat{\zeta}_1(s_{0,0}, s_{0,1}, s_{0,2} + (\frac{b+a}{2}) \cdot c_0, \hat{s}_0) \cdot \hat{y}_0^{c_0}$
- $\hat{t}_1 \stackrel{?}{=} \hat{\zeta}_2(s_{1,1}, \hat{s}_1) \cdot \hat{y}_1^{c_1}$
- $\hat{y}_0 \stackrel{?}{=} \text{commit}(\hat{y}_0, r_{\hat{y}_0})$
- $\hat{y}_1 \stackrel{?}{=} \text{commit}(\hat{y}_1, r_{\hat{y}_1})$
- $\hat{t}_0 \stackrel{?}{=} \text{commit}(\hat{t}_0, r_{\hat{t}_0})$
- $\hat{t}_1 \stackrel{?}{=} \text{commit}(\hat{t}_1, r_{\hat{t}_1})$

Check whether $\text{isConsistent}(c, \{[c_i]\}, \Gamma^*(n))$ returns true

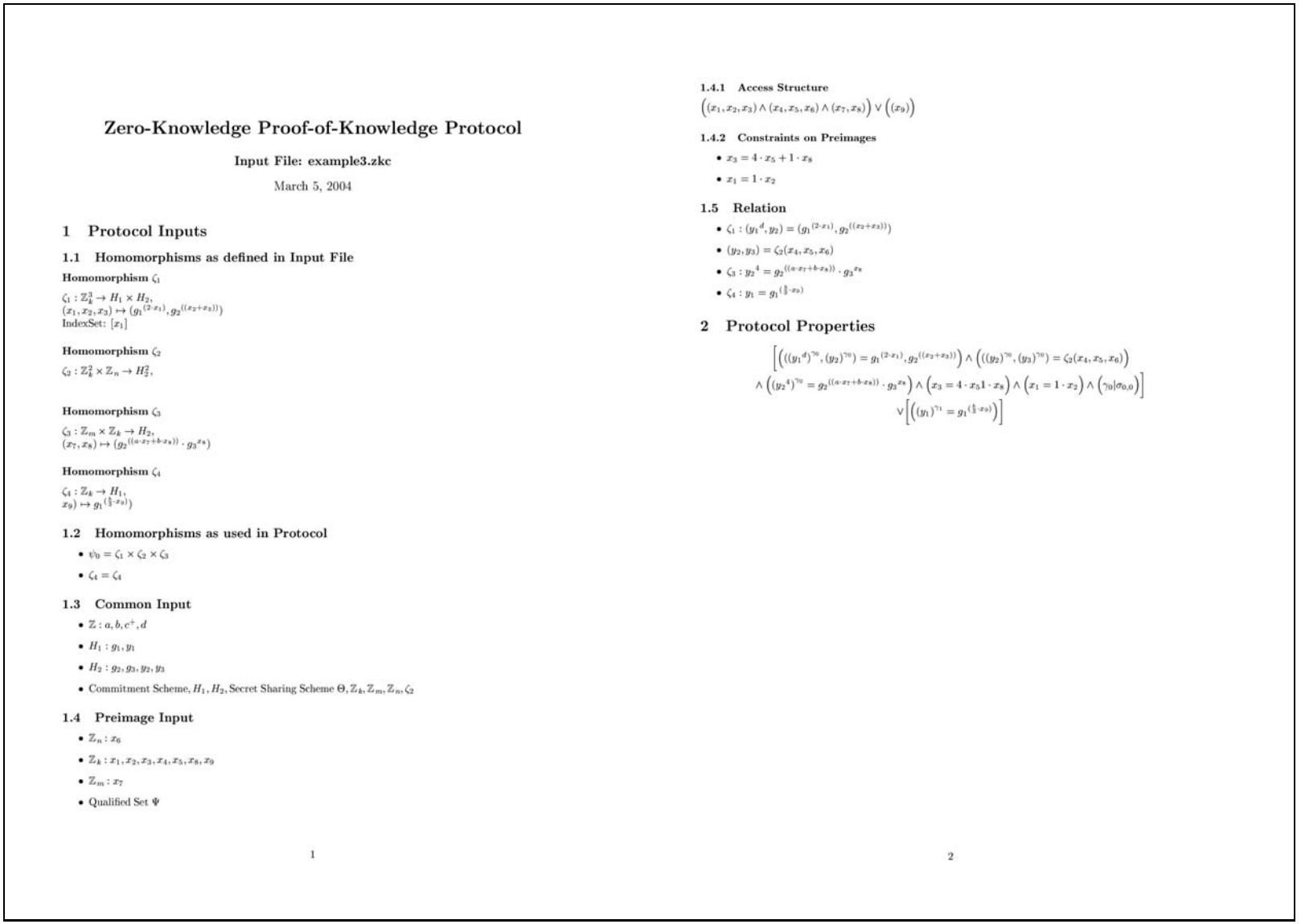
C.3 Concrete Homomorphisms

In example 3 that is shown in figure C.61 some of the homomorphism are specified explicitely as concrete functions. Here are some further remarks:

```
// Declarations
Group (H_[1..2],*), Zm, Zn, Zk;
GroupElement x_[1..9], y_[1..3], g_[1..3];
Homomorphism zeta_[1..4];
IntegerConstant a, b, d, e;

// Assignments
AssignGroupMember(Zk, {x_[1..5], x_[8..9]}),(Zm, x_7), (Zn, x_6);
AssignGroupMember(H_1, {g_1, y_1}), (H_2, {g_[2..3], y_2, y_3});

// Definitions
DefineHomomorphism(zeta_1, (x_[1..3]) |-> (g_1^(2*x_1), g_2^(x_2 + x_3)));
DefineHomomorphism(zeta_2, Zk # Zk # Zn -> H_2 # H_2);
DefineHomomorphism(zeta_3, (x_7, x_8) |-> (g_2^(a*x_7 +b*x_8) * g_3^x_8));
DefineHomomorphism(zeta_4, (x_9) |-> (g_1^((b/3)*x_9)));

//Protocol Specification
SpecifyProtocol[
Relation = [(y_1^d, y_2) = zeta_1(x_1, x_2, x_3)]
&& [(y_2, y_3) = zeta_2(x_4, x_5, x_6)]
&& [(y_2^4) = zeta_3(x_7, x_8)]
|| [(y_1) = zeta_4(x_9)];

Constraints = (x_3 = 4* x_5 + x_8) && (x_1 = x_2);

IndexSet = {x_1};

ParameterNames {
SRSAGenerator = h;
QualifiedSet = Psi;
}

Target = LATEX;
]
```

- In the same input file concrete and abstract definitions of homomorphisms may occur.
- The concrete homomorphism function allows to use group elements as exponents e.g.`g_3^x_8`. This is only possible if the group elements in the exponent belong can be mapped into the integer space. In this case this is possible because they are members of an additive group.

- For the concrete homomorphisms, the groups of the co-domain are deduced from the expressions that are given in the homomorphism definition.
- There are constraints on some preimages. These constraints may only be applied to group elements that belong to the same group. Furthermore the constraints have to be compliant with the chosen access structure. The description of these rules can be found in section 3.3.2. In this example we have only explicit constraints.
- For the execution of the protocol the homomorphisms that are connected through the constraints are combined and form together one single atom.



Zero-Knowledge Proof-of-Knowledge Protocol

Input File: example3.zkc

March 5, 2004

1 Protocol Inputs

1.1 Homomorphisms as defined in Input File

Homomorphism ζ_1

$\zeta_1 : \mathbb{Z}_k^3 \to H_1 \times H_2,$
$(x_1, x_2, x_3) \mapsto ({g_1}^{(2 \cdot x_1)}, {g_2}^{((x_2 + x_3))})$
IndexSet: $[x_1]$

Homomorphism ζ_2

$\zeta_2 : \mathbb{Z}_k^2 \times \mathbb{Z}_n \to H_2^2,$

Homomorphism ζ_3

$\zeta_3 : \mathbb{Z}_m \times \mathbb{Z}_k \to H_2,$
$(x_7, x_8) \mapsto ({g_2}^{((a \cdot x_7 + b \cdot x_8))} \cdot {g_3}^{x_8})$

Homomorphism ζ_4

$\zeta_4 : \mathbb{Z}_k \to H_1,$
$x_9) \mapsto {g_1}^{(\frac{1}{3} \cdot x_9)})$

1.2 Homomorphisms as used in Protocol

- $\psi_0 = \zeta_1 \times \zeta_2 \times \zeta_3$
- $\zeta_4 = \zeta_4$

1.3 Common Input

- $\mathbb{Z} : a, b, c^+, d$
- $H_1 : g_1, y_1$
- $H_2 : g_2, g_3, y_2, y_3$
- Commitment Scheme, H_1, H_2, Secret Sharing Scheme $\Theta, \mathbb{Z}_k, \mathbb{Z}_m, \mathbb{Z}_n, \zeta_2$

1.4 Preimage Input

- $\mathbb{Z}_n : x_6$
- $\mathbb{Z}_k : x_1, x_2, x_3, x_4, x_5, x_8, x_9$
- $\mathbb{Z}_m : x_7$
- Qualified Set Ψ

1.4.1 Access Structure

$\Big((x_1, x_2, x_3) \wedge (x_4, x_5, x_6) \wedge (x_7, x_8)\Big) \vee \Big((x_9)\Big)$

1.4.2 Constraints on Preimages

- $x_3 = 4 \cdot x_5 + 1 \cdot x_8$
- $x_1 = 1 \cdot x_2$

1.5 Relation

- $\zeta_1 : ({y_1}^d, y_2) = ({g_1}^{(2 \cdot x_1)}, {g_2}^{((x_2 + x_3))})$
- $(y_2, y_3) = \zeta_2(x_4, x_5, x_6)$
- $\zeta_3 : {y_2}^4 = {g_2}^{((a \cdot x_7 + b \cdot x_8))} \cdot {g_3}^{x_8}$
- $\zeta_4 : y_1 = {g_1}^{(\frac{1}{3} \cdot x_9)}$

2 Protocol Properties

$$\Big[\Big(({(y_1}^d)^{\gamma_0}, (y_2)^{\gamma_0}) = {g_1}^{(2 \cdot x_1)}, {g_2}^{((x_2 + x_3))}\Big) \wedge \Big(((y_2)^{\gamma_0}, (y_3)^{\gamma_0}) = \zeta_2(x_4, x_5, x_6)\Big)$$
$$\wedge \Big({(y_2}^4)^{\gamma_0} = {g_2}^{((a \cdot x_7 + b \cdot x_8))} \cdot {g_3}^{x_8}\Big) \wedge \Big(x_3 = 4 \cdot x_5 1 \cdot x_8\Big) \wedge \Big(x_1 = 1 \cdot x_2\Big) \wedge \Big(\gamma_0 | \sigma_{0,0}\Big)\Big]$$
$$\vee \Big[\Big((y_1)^{\gamma_1} = {g_1}^{(\frac{1}{3} \cdot x_9)}\Big)\Big]$$

Figure C.61: Generated LaTeX code from Example Input File 3

3 Protocol in verbose Notation

Round 1, Verifier:
$(n, g) := I_{SRSA}(\sigma_s)$.
$\rho_0 \in_U [0, 2^{\sigma_s} \lfloor \frac{n}{4} \rfloor]$
$h_0 := h^{\rho_0}$

n, h_0, h ←

Round 2, Prover:

- if secret $(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8)$ corresponding to ψ_0 is known:
  $r_{0,1} \in_U \mathbb{Z}_k$
  $r_{0,3} \in_U \mathbb{Z}_k$
  $r_{0,4} \in_U \mathbb{Z}_k$
  $r_{0,5} \in_U \mathbb{Z}_n$
  $r_{0,6} \in_U \mathbb{Z}_m$
  $r_{0,7} \in_U \mathbb{Z}_k$
  $r_{0,2} := r_{0,7} \cdot 1 + r_{0,4} \cdot 4$
  $r_{0,0} := r_{0,1} \cdot 1$
  $(t_{0,0}, t_{0,1}) := (g_1^{(2 \cdot r_{0,0})}, g_2^{((r_{0,1} + r_{0,2}))})$
  $(t_{0,2}, t_{0,3}) := \zeta_2(r_{0,3}, r_{0,4}, r_{0,5})$
  $t_{0,4} := (g_2^{((a \cdot r_{0,6} + b \cdot r_{0,7}))} \cdot g_3^{r_{0,7}})$
  $\hat{x}_0 \in_U [0, \lfloor \frac{n}{4} \rfloor]$
  $\hat{y}_0 := \psi_0(x_1, \hat{x}_0)$
  $\hat{r}_0 \in_U [-2^{\sigma_r} c^+ \lfloor \frac{n}{4} \rfloor, 2^{\sigma_r} c^+ \lfloor \frac{n}{4} \rfloor]$
  $\hat{t}_0 := \psi_0(r_{0,0}, \hat{r}_0)$
  $r_{\hat{y}_0} := \text{commitRand}()$
  $r_{\hat{t}_0} := \text{commitRand}()$
  $\hat{y}_0 := \text{commit}(\hat{y}_0, r_{\hat{y}_0})$
  $\hat{t}_0 := \text{commit}(\hat{t}_0, r_{\hat{t}_0})$
- if secret $(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8)$ corresponding to ψ_0 is unknown:
  $s_{0,1} \in_U \mathbb{Z}_k$
  $s_{0,3} \in_U \mathbb{Z}_k$
  $s_{0,4} \in_U \mathbb{Z}_k$
  $s_{0,5} \in_U \mathbb{Z}_n$
  $s_{0,6} \in_U \mathbb{Z}_m$
  $s_{0,7} \in_U \mathbb{Z}_k$
  $s_{0,2} := s_{0,7} \cdot 1 + s_{0,4} \cdot 4$
  $s_{0,0} := s_{0,1} \cdot 1$
  $c_0 \in_U [0, c^+]$
  $(t_{0,0}, t_{0,1}) := (g_1^{(2 \cdot s_{0,0})}, g_2^{((s_{0,1} + s_{0,2}))}) \cdot (y_1{}^{d^{c_0}}, y_2{}^{c_0})$
  $(t_{0,2}, t_{0,3}) := \zeta_2(s_{0,3}, s_{0,4}, s_{0,5}) \cdot (y_2{}^{c_0}, y_3{}^{c_0})$
  $t_{0,4} := (g_2^{((a \cdot s_{0,6} + b \cdot s_{0,7}))} \cdot g_3^{s_{0,7}}) \cdot y_2{}^{4^{c_0}}$
  $\hat{x}_0 \in_U [0, \lfloor \frac{n}{4} \rfloor]$
  $\hat{y}_0 := h^{\hat{x}_0}$
  $\hat{s}_0 \in_U [-2^{\sigma_r} c^+ \lfloor \frac{n}{4} \rfloor, 2^{\sigma_r} c^+ \lfloor \frac{n}{4} \rfloor]$
  $\hat{t}_0 := \psi_0(s_{0,0}, \hat{s}_0) \cdot \hat{y}_0^{c_0}$
  $r_{\hat{y}_0} := \text{commitRand}()$
  $r_{\hat{t}_0} := \text{commitRand}()$
  $\hat{y}_0 := \text{commit}(\hat{y}_0, r_{\hat{y}_0})$
  $\hat{t}_0 := \text{commit}(\hat{t}_0, r_{\hat{t}_0})$

- if secret x_9 corresponding to ζ_4 is known:
  $r_{1,0} \in_U \mathbb{Z}_k$
  $t_{1,0} := (g_1{}^{(\frac{1}{3} \cdot r_{1,0})})$
- if secret x_9 corresponding to ζ_4 is unknown:
  $s_{1,0} \in_U \mathbb{Z}_k$
  $c_1 \in_U [0, c^+]$
  $t_{1,0} := (g_1{}^{(\frac{1}{3} \cdot s_{1,0})}) \cdot y_1{}^{c_1}$

$t_{0,0}, t_{0,1}, t_{0,2}, t_{0,3},$
$t_{0,4}, t_{1,0}, \hat{t}_0, \hat{y}_0$

Round 3, Verifier:
$c \in_U [0, c^+]$

c

Round 4, Prover:
$(c_0, c_1) := \text{complete}(c, \{[c_i]_\Phi\}, \Gamma^*(n))$

- if secret $(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8)$ corresponding to ψ_0 is known:
  $(s_{0,0}, s_{0,1}, s_{0,2}) := (r_{0,0}, r_{0,1}, r_{0,2}) + ((-((x_1, x_2, x_3)))) \cdot c_0$
  $(s_{0,3}, s_{0,4}, s_{0,5}) := (r_{0,3}, r_{0,4}, r_{0,5}) + ((-((x_4, x_5, x_6)))) \cdot c_0$
  $(s_{0,6}, s_{0,7}) := (r_{0,6}, r_{0,7}) + ((-((x_7, x_8)))) \cdot c_0$
  $\hat{s}_0 := \hat{r}_0 + (-(\hat{x}_0)) \cdot c_0$

- if secret x_9 corresponding to ζ_4 is known:
  $s_{1,0} := r_{1,0} + ((-(x_9))) \cdot c_1$

$s_{0,0}, s_{0,1}, s_{0,2}, s_{0,3},$
$s_{0,4}, s_{0,5}, s_{0,6}, s_{0,7},$
$s_{1,0}, \hat{s}_0, c_0, c_1$

Round 5, Verifier:

ρ_0

Round 6, Prover:
Check whether

- $h_0 \stackrel{?}{=} h^{\rho_0}$

$\hat{t}_0, r_{\hat{t}_0}, \hat{y}_0, r_{\hat{y}_0}$

Round 7, Verifier:
Check whether

- $s_{0,2} \stackrel{?}{=} 4 \cdot s_{0,4} + 1 \cdot s_{0,7}$
- $s_{0,0} \stackrel{?}{=} 1 \cdot s_{0,1}$

Check whether

- $(t_{0,0}, t_{0,1}) \stackrel{?}{=} (g_1{}^{(2 \cdot s_{0,0})}, g_2{}^{((s_{0,1} + s_{0,2}))}) \cdot ({y_1}^{c_0}, {y_2}^{c_0})$
- $(t_{0,2}, t_{0,3}) \stackrel{?}{=} \zeta_2(s_{0,3}, s_{0,4}, s_{0,5}) \cdot ({y_2}^{c_0}, {y_3}^{c_0})$
- $t_{0,4} \stackrel{?}{=} (g_2{}^{((a \cdot s_{0,6} + b \cdot s_{0,7}))} \cdot {g_3}^{s_{0,7}}) \cdot {y_2}^{4 c_0}$
- $t_{1,0} \stackrel{?}{=} (g_1{}^{(\frac{b}{3} \cdot s_{1,0})}) \cdot {y_1}^{c_1}$
- $\hat{t}_0 \stackrel{?}{=} \hat{\psi}_0(s_{0,0}, \hat{s}_0) \cdot \hat{y}_0^{c_0}$
- $\bar{y}_0 \stackrel{?}{=} \text{commit}(\hat{y}_0, r_{\hat{y}_0})$
- $\bar{t}_0 \stackrel{?}{=} \text{commit}(\hat{t}_0, r_{\hat{t}_0})$

Check whether $\text{isConsistent}(c, \{[c_i]\}, \Gamma^*(n))$ returns true

C.4 Example with implicit constraints

Example 4 shows the specification of an instance of the Σ protocol . The LaTeX output of this file is shown in figure C.62.

```
// Declarations
Group (G_[1..4],+), (H, *);
GroupElement x_[1..8], y_[1..4];
Homomorphism zeta_[1..4];

// Assignments
AssignGroupMember(G_1, {x_[1..3]}), (G_2, x_4), (G_3, {x_[5..7]}), (G_4, x_8);
AssignGroupMember(H, {y_[1..4]});

// Definitions
DefineHomomorphism(zeta_1, G_1 # G_1 -> H);
DefineHomomorphism(zeta_2, G_1 # G_2 -> H);
DefineHomomorphism(zeta_3, G_3 # G_3 # G_3 -> H);
DefineHomomorphism(zeta_4, G_1 # G_4 # G_1 -> H);

// Protocol Specification
SpecifyProtocol[
Relation [
CommonInput = {(zeta_1,(y_1)), (zeta_2,(y_2)),(zeta_3,(y_3)), (zeta_4,(y_4))};
PreimageInput = {(x_1, x_2), (x_1, x_4), (x_5, x_6, x_7), (x_2, x_8, x_3)};
AccessStructure = {{zeta_1, zeta_2, zeta_4}, {zeta_3}};
]

Target = LATEX;
Layout = COMPACT;
]
```

Some remarks on this example:

- In this example the relation is written in enumerative description. An access structure is specified in this case in naming the qualified sets.
- Even though no explicit constraints are specified, there are implicit constraints as the preimages `x_1` and `x_2` are used in several homomorphisms. This has exactly the same effect as if there would have been explicit constraints and therefore have to be fit together with the access structure. These homomorphisms form together a new atom.

Zero-Knowledge Proof-of-Knowledge Protocol

Input File: example4.zkc

March 5, 2004

1 Protocol Inputs

1.1 Homomorphisms as defined in Input File

Homomorphism ζ_1

$\zeta_1 : G_1^2 \to H,$

Homomorphism ζ_2

$\zeta_2 : G_1 \times G_2 \to H,$

Homomorphism ζ_3

$\zeta_3 : G_3^3 \to H,$

Homomorphism ζ_4

$\zeta_4 : G_1 \times G_4 \times G_1 \to H,$

1.2 Homomorphisms as used in Protocol

- $\psi_0 = \zeta_1 \times \zeta_2 \times \zeta_4$
- $\zeta_3 = \zeta_3$

1.3 Common Input

- $H : y_1, y_2, y_3, y_4$
- $\mathbb{Z} : e^+$
- G_1, G_2, G_3, G_4, H, Secret Sharing Scheme $\Theta, \zeta_1, \zeta_2, \zeta_3, \zeta_4$

1.4 Preimage Input

- $G_4 : x_8$
- $G_1 : x_1, x_2, x_3$
- $G_2 : x_5, x_6, x_7$
- $G_3 : x_4$
- Qualified Set A

1.4.1 Access Structure

$$\Big((x_1, x_2) \wedge (x_1, x_4) \wedge (x_2, x_8, x_3)\Big) \vee \Big((x_5, x_6, x_7)\Big)$$

1.4.2 Constraints on Preimages

- $x_{1\zeta_2} = 1 \cdot x_{1\zeta_1}$
- $x_{2\zeta_4} = 1 \cdot x_{2\zeta_1}$

1.5 Relation

- $y_1 = \zeta_1(x_1, x_2)$
- $y_2 = \zeta_2(x_1, x_4)$
- $y_3 = \zeta_3(x_5, x_6, x_7)$
- $y_4 = \zeta_4(x_2, x_8, x_3)$

2 Protocol Properties

$$\Big[\Big((y_1)^{\gamma_0} = \zeta_1(x_1, x_2)\Big) \wedge \Big((y_2)^{\gamma_0} = \zeta_4(x_1, x_4)\Big) \wedge \Big((y_4)^{\gamma_0} = \zeta_4(x_2, x_8, x_3)\Big) \wedge \Big(x_{1\zeta_2} = 1 \cdot x_{1\zeta_1}\Big) \wedge \Big(x_{2\zeta_4} = 1 \cdot x_{2\zeta_1}\Big)\Big] \vee \Big[\Big((y_3)^{\gamma_1} = \zeta_3(x_5, x_6, x_7)\Big)\Big]$$

Figure C.62: Generated LaTeX code from Example Input File 4

3 Protocol in compact Notation

Round 1, Prover:
Iterate over homomorphims ψ_0, ζ_3 :

- if secret x_{ζ_i} is known:
  $r_{\zeta_i} \in_U G^{q'}_{\zeta_i}$
  $t_{\zeta_i} := \zeta_i(r_{\zeta_i})$
- if secret x_{ζ_i} is unknown:
  $s_{\zeta_i} \in_U G^{q'}_{\zeta_i}$
  $c_i \in_U [0, c^+]$
  $t_{\zeta_i} := \zeta_i(s_{\zeta_i}) \cdot y^{c_i}_{\zeta_i}$

t_{ζ_i} →

Round 2, Verifier:
$c \in_U [0, c^+]$

← c

Round 3, Prover:
$(c_0, c_1) := \text{complete}(c, \{[c_i]_{\bar{A}}\}, \Gamma^*(n))$
Iterate over homomorphims ψ_0, ζ_3 :

- if secret x_{ζ_i} is known:
  $s_{\zeta_i} := r_{\zeta_i} + (-x_{\zeta_i}) \cdot c_i$

s_{ζ_i}, c_i →

Round 4, Verifier:
Check whether

- $s_{0,2} \stackrel{?}{=} 1 \cdot s_{0,0}$
- $s_{0,4} \stackrel{?}{=} 1 \cdot s_{0,1}$

Check for each homomorphism ψ_0, ζ_3 whether

- $t_{\zeta_i} \stackrel{?}{=} \zeta_i(s_{\zeta_i}) \cdot y^{c_i}_{\zeta_i}$

Check whether $\text{isConsistent}(c, \{[c_i]\}, \Gamma^*(n))$ returns true