DISS. ETH NO. 20513

A Framework for Modelling and Testing of Security Policies

A dissertation submitted to

ETH ZURICH

for the degree of

Doctor of Sciences

presented by

Lukas Alexander Brügger

Dipl. Informatik Ingenieur ETH

born 14.06.1982

citizen of Plaffeien

accepted on the recommendation of

Prof. Dr. David Basin
Prof. Dr. Peter Müller
Prof. Dr. Burkhart Wolff

2012

Abstract

In this thesis, we present a uniform framework for modelling security policies and show how this framework is used for model-based conformance testing of systems implementing these policies. The framework can be used to model a wide range of different kinds of security policies. It also provides support for modelling dynamic system behaviour for modelling stateful policies. The framework supports both unit and sequence tests, and includes techniques for automatic test execution.
We show how to model large-scale policies and systems, and present techniques that allow for an efficient and effective testing of these systems. Our contributions include support for formally verified policy transformation procedures, allowing one to tame inherent state-space explosions in test case generation for security policies.
The framework is implemented in a theorem proving environment and has a rigorous formal foundation. We provide powerful techniques to reason about policies and support for a tighter integration of tests and proofs.
We provide evidence of the strength of our framework by instantiating it with two large-scale case studies: one is in the area of firewalls and networks, and the other in the area of access to electronic health care records.

Zusammenfassung

In dieser Arbeit präsentieren wir ein einheitliches Framework für die Modellierung von Sicherheitsrichtlinien und zeigen, wie dieses zur model-basierten Konformitätsprüfung dieser Systeme benutzt werden kann. Das Framework bietet die Möglichkeit, eine Vielzahl sehr unterschiedlicher Sicherheitsrichtlinien modellieren zu können. Es bietet auch Unterstützung für Modellierung dynamischen Systemverhaltens, was für die Modellierung von zustandsbehafteten Sicherheitsrichtlinien benutzt werden kann. Das unterstützt sowohl Unit wie auch Sequenz-Tests. Ausserdem bietet es auch Techniken für die automatische Testdurchführung an.
Wir zeigen, wie auch sehr grosse Sicherheitsrichtlinien und Systeme modelliert werden können, und präsentieren Techniken für ein effizientes und effektives Testen dieser Systeme. Unsere Beiträge beinhalten das Konzept von formal verifizierten Transformationen von Sicherheitsrichtlinien, die es ermöglichen, die inhärenten Zustandsexplosionen zu vermindern.
Das Framework basiert auf einem Theorembeweiser und hat eine starke formale Grundlage. Wir bieten leistungsstarke Techniken um Sicherheitsrichtlinien zu analysieren, und bieten Unterstützung für eine engere Integration von Tests und Beweisen.
Wir belegen die Stärke unseres Framework durch die Instanziierung mit zwei gross angelegten Fallstudien: Die eine ist im Bereich von Firewalls und Netzwerken, die andere im Bereich der Zugriffskontrolle zu elektronischen Patientenakten.

Acknowledgements

This dissertation would not have been possible without the great support that I received from many colleagues, friends, and family members.
Most importantly, I would like to thank Achim Brucker and Burkhart Wolff. They have introduced me to the fascinating world of research many years ago, and have been great colleagues ever since. Most of the work presented in this thesis was created in close collaboration with both of them.
Second, I thank David Basin for giving me the opportunity of pursuing my dissertation in this great research environment, for his beneficial scientific advices, comments, feedback, and his generous support.
I would also like to thank all present and past members of the Institute of Information Security at ETH Zurich, who all contributed to a very pleasant working environment. Especially I thank my office mates Matus Harvan, Achim Brucker, Mohammad Torabi Dashti, Simone Frau, and Farhad Mehta for their good company.
The work presented in this thesis has been developed in close collaboration with British Telecom. I would like to thank all the countless number of BT people, especially Paul Kearney, who all provided me with very valuable information, feedback, and ideas.
Last but not least, I would like to thank my parents, my two sisters, and all my friends for their support during the last couple of years.

Contents

1 Introduction

1.1 Motivation

Computer-based systems are becoming increasingly paramount in today's world. Furthermore, these systems are increasingly interconnected and allow access to sensitive data from various locations. Examples of such systems include centrally stored health care data, remote access to services for company employees, and many others. Besides the large advantages this might lead to in the areas of quality, efficiency, and others, the associated security risks are far from negligible. Accounts of unauthorised access to systems and data breaches are omnipresent and the situation clearly needs to be improved.

While the systems in question are generally quite distinct from each other, one common characteristic can be observed in many of them: the access to sensitive parts of the system (for example to system functionality or to confidential data) is governed by one or more *security policies*. A security policy is a description of how the systems need to be configured in order to deny unauthorised and allow authorised accesses. While security policies as such are a relatively well-studied domain in the literature, the corresponding system configurations are often surprisingly hard to get right. Reasons include the fact that policies tend to grow very large and complex, different policies of various kinds need to be integrated, and the systems themselves consist of a large number of often distributed sub-systems which all must be configured accordingly.

In fact, for many systems, the likelihood of a security vulnerability arising from misconfiguration is much greater than from a bug in the software itself. For example, consider the following quotation:

> "NSA found that inappropriate or incorrect security configurations (most often caused by configuration errors at the local base level) were responsible for 80 percent of Air Force vulnerabilities." [cis08, p. 55]

However, the policies and their correct configurations and implementations are a central point of modern security architectures, be it a firewall policy restricting unwanted network traffic, be it an access control policy specifying that only legitimate users may access a service, be it an information governance policy specifying that users are only presented the data they are required to see in order to perform their job.

Unfortunately a full verification of the conformance of a system to a security policy is often not feasible. Testing on the other hand can play an important role towards improving the quality of the systems. However, testing is accompanied with two large problems: the first is how to choose a reasonable set of test data. Any approach which is based on random testing or on manual creation of test cases is unlikely to provide good results in the security testing area. The second problem is the costs and time required to execute the tests on the system. This is often considered as too large, limiting the effects of testing towards a considerable improvement of the situation.

Most traditional approaches to testing use a white-box approach. The test cases are generated from the code to be tested, and the quality of a test set is measured by code coverage metrics. In the domain of validating conformance of a set of systems implementing a security policy, this approach is usually not possible, as the source code of the systems in question is often not completely available. The reasons include that often the systems are configured by entities different from those that implemented them, and that the policies usually have to be implemented by a set of systems possibly coming from different sources.

A common problem in policy testing is the fact that policies tend to grow very large. Depending on which coverage criterion is applied, this might lead to significant scalability problems and drive the costs of the tests upwards. Applying a lower coverage on the other hand, might lead to important parts of the specification not being validated.

In this thesis, we propose a solution to these problems by applying model-based testing techniques to the area of model-based conformance testing of security policies. This testing approach, being black-box and specification based, is best suited for application domains where we need to validate that a range of diverse and potentially at least partially unknown systems conform in their common execution to a given (high-level) policy description. The approach we propose allows for a uniform processing of a wide range of different kinds of policies and has a rigorous formal foundation. The formal foundation also gives rise to a technique called verified policy transformations, providing a partial solution to the scalability problem of security policy testing.

1.2 Main Ideas

The goal of the work presented in this thesis is to provide a framework for the model-based conformance testing of a wide range of security policies, encompassing the complete testing process. To achieve this, we have developed the following:

- **The Unified Policy Framework.** We present a formal framework which allows for modelling a wide range of security policies, called the Unified Policy Framework (UPF). The framework is based on the notion of combination of sub-policies, and also allows for the specification of dynamic system behaviour. Due to its in-

tegration into a theorem proving environment, it provides sophisticated machine-supported reasoning techniques.

- **Techniques for conformance testing of security policies.** We provide the means and the techniques to test systems for conformance to policies specified in the modelling framework. We support both unit and sequence tests, and also provide techniques for automatic test execution.

- **Transformation procedures for policies.** We present a technique allowing for provably correct testability transformations for security policies, which allow for a considerably more efficient testing process.

- **Instantiation of the framework in the domain of access to health care records.** We present an instantiation of the policy modelling framework with the information governance policies of the National Programme for Information Technology (NPfIT) in the English NHS. As these policies, drawn from a large-scale real-world example, are very complex and diverse, they are a proof of concept for the strength of the modelling framework. We also report how these models can be used for testing systems having to conform to these policies, accompanied with a description of a prototypical web-service based test execution framework.

- **Instantiation of the framework in the firewall domain.** We present a large-scale case study by instantiating the framework with firewall and network policies. We show how different kinds of policies can be specified and be used for test case generation, including a thorough empirical evaluation of the test case generation process. Furthermore, we present a domain-specific testing tool for automatic execution of these tests on a network.

1.3 Contributions

The main contribution of this thesis is a novel framework for the model-based conformance testing of security policies with the following characteristics:

- While there exist several frameworks for modelling security policies, and also for model-based test generation, our framework encompasses the complete testing process. Specifically, it provides a modelling language, algorithms for the generation of abstract and concrete test data from the model, and several techniques for automatic test execution. Explicit test specifications enable an interactive test selection approach.

- The policy modelling language uses a slightly non-standard representation of the fundamental policy concept. This choice leads to a couple of advantages regarding modelling and testing. The language is largely based on policy combination

operators, leading to a modular modelling approach. Furthermore, it is tightly integrated with the possibility to model stateful systems, allowing one to test systems for conformance to stateful policies. Currently, no such policy modelling language exists.

- The language is based on higher-order logics and the whole framework is integrated into a theorem-proving environment. There is a rigorous formal foundation of the complete process, and explicit test hypotheses enable a close integration with verification approaches. Furthermore, we provide powerful machine-supported policy reasoning techniques.

- We provide a novel approach to the scalability problem inherent in testing of security policies of realistic size. This approach is based on the concept of testability transformations, which for the first time is applied to the domain of security policies. As a particular contribution, the provided transformations are all verified to keep the semantics of the specification inside the same environment as the rest of the framework.

- The framework is applicable to a wide range of security policies from a large number of different domain. Most current approaches focus on one particular domain.

- We provide evidence of the strength of the modelling framework by instantiating it with policies of a large-scale industrial system from the health-care domain. In this instantiation, the systems have to conform to a range of diverse and large policies.

- We provide evidence of the practical applicability of the framework by instantiating it with policies of the firewall and networking domain, including a domain-specific test execution tool. The framework has been applied to a number of real firewalls.

For most of the points mentioned above, several approaches exist that provide some parts of the specific contribution. For the first time however, the framework presented in this thesis integrates all of these contributions, providing a generic framework for model-based conformance testing of security policies from a wide range of different domains, encompassing the complete testing process with a strong formal foundation. As a particular contribution, we provide a partial solution to the scalability problem inherent in testing industrial-scale security policies.

1.4 Overview

The thesis is organised as follows. In the next chapter, we provide the necessary background on the foundations of this thesis, namely the concepts of formal testing and of security policies. Furthermore we present the tools we use for the implementation of our

framework. Chapter 3 presents the Unified Policy Framework (UPF), a framework for the modelling of security policies. The following chapter 4 presents the techniques to use the models of the UPF for conformance testing, including the concept of testability transformations for policies. Chapter 5 presents the instantiation of the framework with the domain of access to health care records, and chapter 6 with the domain of firewall policies. After this, chapter 7 presents related work, before we conclude and present some future research ideas in chapter 8.

All the definitions, functions, theorems, and proofs presented in this thesis have been formalised and checked in the theorem prover Isabelle/HOL. We just use some minor syntactical simplification for the sake of presentation.

Parts of the work presented in this thesis has already been published in parts before. In particular, [BBW08a, BW07] introduce the firewall case study, [BBKW11] the health care case study, and [BBKW10] a transformation procedure for firewall policies.

2 Foundations and Background

In this chapter, we provide the necessary background on the thesis. Namely, we introduce the concepts of security policies, of formal testing, and finally of their combination: the model-based conformance testing of security policies. The presentation is accompanied with a description of the underlying systems used in our implementation: Isabelle/HOL and HOL-TestGen.

2.1 Security Policies

The main objects under consideration in this thesis are *security policies*. Security policies are a cornerstone for ensuring security in any modern IT system. The term is very broad and somewhat overloaded in the literature. In essence, a security policy specifies who (e.g. users, processes, network packets) may do what (e.g. write an file, read an object, pass through a firewall, start a process) under which circumstances (e.g. during daytime only, if originating in the correct network, if being a member of a correct role). The policy is the specification which has to be used to configure an access control system to implement the intended behaviour. This thesis deals with the question how to validate conformance of the implementation to the security policy specification.

More formally, the term *security policy* can be defined as *"defining the (high-level) rules according to which access control must be regulated"* [SV01], whereas *access control* is defined as requiring that *"every access to a system and its resources be controlled and that all and only authorised accesses can take place"* [SV01].

Sometimes, instead of the term security policy the terms authorisation policy and access control policy are used, where the last two often have a slightly more narrow meaning. As an example, in [BH11] the authors define access control as the *"methods or mechanisms that decide whether requests to access some resource should be granted or denied"*. The term security policy on the other hand is often used in a much wider meaning, for example policies determining the physical mechanisms of a protected building.

Research into the areas of security policies has started in the 1970s (for example in [But74]) when people were interested in the protection of resources in multi-user operating systems. Since then, a wide range of access control models has been proposed. When applying a narrow definition to the term security policy, that is the enforcement of ac-

cess of subjects to objects, the following two categories of access control models are most relevant:

Discretionary Access Control: In discretionary access control models, users are the owners of objects and have the capabilities to grant and revoke rights to these objects to other users at their discretion. The model is usually represented in the form of an access matrix (see [But74] and [HRU76]) with various implementation variants (for example access control lists and capability lists). One of the main problems of discretionary access control models is the fact that they are relatively hard to administer and maintain.

Mandatory Access Control: In mandatory access control, the users have no influence on the access rights, rather they are imposed by a central authority. The most important kind of mandatory access control models are multi-level security models, where users and objects are being classified in different categories. Several principles can be used to specify what users in some classifications may do with objects from other classifications. The prime examples of multilevel security models are the Bell-LaPadula model [BL96] and the Biba model [Bib77]. The main disadvantage of mandatory access control models is the fact that they are often too rigid and allow little flexibility.

Role-based Access Control (RBAC) is a proposal to overcome the two main disadvantages of discretionary and mandatory access control models. The standard model of RBAC has been introduced by [SCFY96], and in the meantime many different variants have been proposed (for example [MA01, BBF01, SBM99]). At their core they all have the concept of an additional indirection as users are being allocated to roles, and roles to permissions. A user's permissions are determined by the permissions being granted to the roles the user possesses. Variants of RBAC include concepts like role delegation, parametrised roles, and role hierarchies. RBAC has been quite successful due to the fact that it scales very well and makes administration of an access control system much simpler.

In recent years, those standard access control models have been extended in various ways, for example to integrate temporal concepts, obligations, and further support for administrative policies. Furthermore, several approaches for access control models using logic-based languages (e.g. [BN10]) have been proposed.

In addition to the above presented policy models which are mainly used for authorisation-based policies, further models have been introduced for domain-specific scenarios. One example is the domain of privacy policies with the policy modelling languages P3P, EPAL, and others [KS02, BDMN06, CLMR02]. Essentially, those models are extensions of the authorisation policies presented above. As another domain-specific example, firewall and network policies are often viewed as something quite distinct, but are actually conceptually very similar to more standard access control policies.

The aforementioned policy models, be it the standard or the domain-specific ones, share many similarities from a conceptual point of view. In the end, they can all be represented by a function taking some kind of input and possibly state information, and providing as part of the output a decision, denoting if the corresponding access request should succeed or fail. In this thesis, we will provide a framework for security policy modelling taking advantage of this fact, where all these policy models can be represented.

2.2 Formal Testing

Modern software systems are of such an enormous complexity, that only rarely they are constructed correctly from the start. For this reason, testing of software is absolutely necessary. The disadvantage of testing is the fact that it is very labour- and cost-intensive. There are estimates that on average testing consumes 30-50% of the total software development costs [Tre08], but still software quality has in general not yet reached the desired level. Furthermore, Dijkstra's famous verdict seems to be giving a quite negative view on the value of testing as well:

> Program testing can be used to show the presence of bugs, but never to show their absence! [Dij72]

One way to improve the software engineering process, contributing in reduced testing efforts required, are *Formal Methods*, that is to use mathematical techniques in the software engineering process (for example proofs of correctness of a software). While they do have the potential for large quality improvements, formal methods alone are not sufficient and can only be complementary to testing activities. The reasons have already been described in 1975 by Goodenough [GG75], where he states that proofs only provide assurance of correctness if the environment can be axiomatised completely, which is unrealistic. Furthermore, the systems to test might at-least partially only be provided as a black-box and modelling errors are always possible.

For the reasons mentioned above, both formal proofs and testing are insufficient for providing assurance on their own and need to be applied in a complementary manner. One approach to bring the two aspects closer together, but that also has the potential of heightening the quality of testing, is the concept of *formal testing*. The idea of formal testing is to base testing methods on formal specifications and provide a solid foundation on its theory, such that the following quote of Goodenough is no longer valid:

> We know less about the theory of testing, which we do often, than about the theory of program proving, which we do seldom. [GG75].

Formal testing has become more mainstream in the 1990s, starting with research in the area of testing communication protocols. Gaudel [Gau95], Dick and Faivre [DF93] and others extended this work towards general theories of testing. In essence, they

reason about different test methods and most notably about the corresponding test selection process, that is "the process of dividing the input domain of the program into subdomains and require the test set to include at least one element from each subdomain" [Gau95]. They define notions of completeness, soundness, and exhaustiveness of a test set and provide definitions for explicit test (or selection) hypotheses that provide a formal correspondence between tests and proofs by making explicit under which conditions the execution of a successful test set will imply correctness of the system under test. The choice of the (concrete) test hypotheses implies the test selection criteria and determines the test strategy. As such, test hypotheses are closely related to the concept of fault models.

The two most important test hypotheses regarding our framework are:

- **Uniformity Hypothesis:** This paraphrases above criteria: if the test passes for one element of a subdomain, it passes for all elements of that subdomain
- **Regularity Hypothesis:** Given a size function for a type, if the tests pass for all elements up to a given size determined by depth of the test selection, they will pass for all elements of a larger size.

A considerable advantage of formal testing over other forms of testing, is the fact that the specification cannot only be used as a means for test selection, but also serves as a *test oracle*. That is, we may use the specification as a means for verifying the test results. However, in general, the specification is not on the same layer of abstraction as the implementation, there is no direct bijection between elements returned by the specification and by the implementation. Different approaches have been proposed to overcome this problem, but in general a user-supplied abstraction function is required.

A further advantage of formal testing is the fact that formal testing allows for large parts of the testing process to be automated. This does not only hold for the test selection process as outlined above, but also for large parts of the test execution.

One important subclass of formal testing is *model-based testing*, which has recently gained quite a lot of attention, and is also starting to be used in industry on a large scale (e.g. see [VCG+08]). Model-based testing is a general term that specifies a formal testing approach that bases common testing tasks such as test case generation on a model of the application under test [EFW01].

In general, model-based testing compares the actual behaviour of a given system under test (SUT) with its desired behaviour as described in its specification (a *model*). It involves the automated derivation of a set of test data from the specification, each of which contains a set of inputs and/or description of a system state, and criteria for classifying whether the observed system output complies with the specification. The test inputs are applied to the system under test (SUT) and the results are classified (for example as pass or fail). If all tests (or a sufficient number of them) pass, the SUT is judged to be compliant with its specification.

Please note that the distinction between the terms formal testing and model-based testing is not always very clear. We use the same definition as Tretmans [Tre08], who defines model-based testing as "*formal, specification based, active, black-box, functionality testing*", where the terms in the definition have the following meaning:

formal: as outlined above: the model of the system is given in some formal language, and a formal process is used to generate the tests

specification based: The tests are based on a specification of the system. This specification is assumed to be a valid description of the system under test.

active: The tests are executed actively, that is the tester introduces test data and observes the results. This is in contrast to passive testing where the tester only observes the normal execution of the system.

black-box: We only have the specification and the public interfaces of the system under test available and do not know anything about its internal details.

functionality: We focus on functionality testing, and not for example performance, or usability testing.

A slightly broader, but closely related definition is given by Utting and Legeard:

Definition 2.2.1 *Model-based testing is the automation of the design of black-box tests. [UL06]*

Note that by both of these definitions, formal testing is a superclass of model-based testing.

2.3 Theorem-prover based Testing

There are many different approaches to model-based testing (see also chapter 7). In this section, we present one particular approach on which we base the work presented in this thesis: *theorem-prover based testing*.

In short, *theorem provers* are used to prove statements given in a formal language. There are essentially two categories: automated theorem provers which allow one to automatically prove mathematical theorems and interactive theorem provers that only offer a limited degree of automation and require the presence of a proof engineer who guides the proving process.

Reflecting their main purpose, theorem provers are often also called *proof assistants*. A proof conducted in, or found by, a theorem prover is formally checked by a computer and

thus guaranteed to be correct, unlike a proof written in natural language. This holds under the assumption of correctness of the implementation of the theorem prover (resp. of its small, trusted kernel only) and the underlying logic.

Among others, theorem provers have successfully been used in the following areas:

- Hardware verification
- Formalisation of large parts of mathematics
- Proving of very hard mathematical theorems
- Formalisation of programming languages
- Proving properties of software

Theorem-prover based testing is an approach which integrates the testing process into a theorem proving environment. In more detail, the model of the system, possibly together with a model of the environment and some constraints on the input space which is used for automating the test selection process is given in the formal language of the theorem prover. This model is called the test specification. Furthermore, also the test selection process from the test specification is a formally checked procedure implemented in the prover. Besides only creating the test data from the model, the procedure will also generate the test hypotheses as a formal description of remaining proof obligations that need to be verified in order to prove correctness of the test specification. In other words, testing is seen as an approximation to verification, with the test hypotheses being the link between the two concepts. As such, this approach also sees test hypotheses as an alternative to test adequacy criteria. Instead of stating when we have tested enough, the focus is on what remains to be proved (see also [BW12]). Thus in summary, the theorem-prover based testing approach aims for a tight integration of verification and testing. We have explored some of the connections of tests and proofs in [BBW08b].

As another advantage, any test case generation technique depends on symbolic computation and constraint-solving techniques. The limitations of these techniques therefore represent limits for model-based testing techniques as a whole. Theorem-prover based testing systems contain powerful automated and interactive proof methods for constraint resolution and directly provide powerful modelling languages. In the following we will present one particular approach to theorem-prover based testing: namely the HOL-TestGen tool which is implemented on top of the theorem prover Isabelle/HOL.

2.3.1 Isabelle and Higher-order Logic

Isabelle [NPW02] is a *generic* theorem prover. New object logics can be introduced by specifying their syntax and natural deduction inference rules. Among other logics, Isabelle supports first-order logic, Zermelo-Fraenkel set theory and an instance for

Church's Higher-order Logic (HOL).

Isabelle's inference rules are based on the built-in meta-level implication $\_ \Longrightarrow \_$. Using this, one can form constructs like $A_1 \Longrightarrow \cdots \Longrightarrow A_n \Longrightarrow A_{n+1}$, which are viewed as a *rule* of the form "from assumptions A_1 to A_n, infer conclusion A_{n+1}" and which is written in Isabelle as

$$[\![A_1; \ldots; A_n]\!] \Longrightarrow A_{n+1} \qquad \text{or, in mathematical notation,} \qquad \frac{A_1 \quad \cdots \quad A_n}{A_{n+1}}$$

The built-in meta-level quantification $\bigwedge x.\ Px$ captures the usual side-constraints "x must not occur free in the assumptions" for quantifier rules; meta-quantified variables can be considered as "fresh" free variables. Meta-level quantification leads to a generalisation of Horn-clauses of the form:

$$\bigwedge x_1, \ldots, x_m.\ [\![A_1; \ldots; A_n]\!] \Longrightarrow A_{n+1}\,.$$

Isabelle supports forward- and backward reasoning on rules. For backward-reasoning, a *proof state* can be initialised and further transformed into other proof states. For example, a proof of ϕ, using the Isar [Wen02] language, might look as follows in Isabelle:

```
lemma label:  φ
  apply(case_tac)
  apply(simp_all)
done
```

This proof script instructs Isabelle to prove ϕ by case distinction followed by a simplification of the resulting proof state. Such a proof state is an implicitly conjoint sequence of generalised Horn-clauses (called *subgoals*) $\phi_1, \ldots, \phi_n$ and a *goal* ϕ. Proof states are usually denoted by:

$$\begin{array}{rl} \text{label}: & \phi \\ 1. & \phi_1 \\ & \vdots \\ n. & \phi_n \end{array}$$

Subgoals and goals may be extracted from the proof state into theorems of the form $[\![\phi_1; \ldots; \phi_n]\!] \Longrightarrow \phi$ at any time; this mechanism helps to generate test theorems. Further, Isabelle supports meta-variables, which can be seen as "holes in a term" that can still be substituted. Meta-variables are instantiated by Isabelle's built-in higher-order unification.

Higher-order logic (HOL) [Chu40, And02] is a classical logic based on a simple type system. It provides the usual logical connectives like $\_ \wedge \_$, $\_ \rightarrow \_$, $\neg \_$ as well as the object-logical quantifiers $\forall x.\ P\,x$ and $\exists x.\ P\,x$; in contrast to first-order logic, quantifiers my range over arbitrary types, including total functions $f :: \alpha \Rightarrow \beta$. HOL is based on extensional equality $\_ = \_ :: \alpha \Rightarrow \alpha \Rightarrow$ bool. HOL is more expressive than first-order logic, induction schemes for example can be expressed inside the logic. Being based on some polymorphically typed λ-calculus, HOL can be viewed as a combination of a programming language like Standard Meta Language (SML) or Haskell, and a specification language providing powerful logical quantifiers ranging over elementary and function types.

Isabelle/HOL is a logical embedding of HOL into Isabelle. The (original) simple type system underlying HOL has been extended by Hindley/Milner style polymorphism with type-classes similar to Haskell. While Isabelle/HOL is usually seen as "proof assistant," we use it as symbolic computation environment. Implementations on top of Isabelle/HOL can re-use existing powerful deduction mechanisms such as higher-order resolution, tableaux-based reasoners, rewriting procedures, Presburger Arithmetic, and via various integration mechanisms also external provers such as Vampire and the Satisfiability Modulo Theories (SMT)-solver Z3.

Isabelle/HOL offers support for a particular methodology to extend given theories in a logically safe way, called *conservative* extensions. If a theory-extension is *conservative*, the extended theory is consistent provided that the original theory was consistent. Conservative extensions in Isabelle can be *constant definitions*, *type definitions*, *datatype definitions*, *primitive recursive definitions* and *well-founded recursive definitions*.

As an example, typed sets are built in the Isabelle libraries conservatively on top of the kernel of HOL as functions to bool; consequently, the constant definitions for set comprehension and membership are as follows:

```
type_synonym α set = α ⇒bool

definition Collect :: (α ⇒bool) ⇒ α set where −− comprehension
 Collect P = P

definition member :: α ⇒α set ⇒bool where −− membership
 member x A = A x
```

Note that Greek characters denote type variables. Thus these definitions can be used for a set of any type. Isabelle's syntax engine is instructed to accept the notation $\{x \mid P\}$ for $Collect\ \lambda x.\ P$ and the notation $x \in A$ for $member\ x\ A$. Constant definitions are axioms that introduce a fresh constant symbol by some closed, non-recursive expressions; this type of axiom is logically safe (that is it can not introduce additional inconsistencies into the theory), since it works like an abbreviation. The syntactic side-conditions of the axiom are mechanically checked. It is straightforward to express the usual operations

on sets like $\_ \cup \_, \_ \cap \_ :: \alpha\,\text{set} \Rightarrow \alpha\,\text{set} \Rightarrow \alpha\,\text{set}$ as conservative extensions, too, while the rules of typed set-theory are derived by proofs from these definitions.

Similarly, a logical compiler is invoked for the following statements introducing the types option[1] and list:

datatype α option $= \bot \mid \lfloor\alpha\rfloor$

datatype α list =
  Nil (" [] ")
| Cons α "α list" (infixr "#" 65)

Here, [] and $a\#l$ are an alternative syntax for Nil and Cons $a\ l$; moreover, $[a, b, c]$ is defined as alternative syntax for $a\#b\#c\#[]$. These (recursive) statements are internally represented by type and constant definitions. Besides the *constructors* $\bot$, $\lfloor\rfloor$, [] and Cons, there is the match-operation case x of $\bot \Rightarrow F \mid \lfloor a\rfloor \Rightarrow G\ a$ respectively case x of $[] \Rightarrow F \mid \text{Cons}\ a\ r \Rightarrow G\ a\ r$. From the internal definitions (not shown here) a number of properties are automatically derived. Here we show the case for lists:

$$(\text{case}\ []\ \text{of}\ [] \Rightarrow F \mid (a\#r) \Rightarrow G\ a\ r) = F$$

$$(\text{case}\ b\#t\ \text{of}\ [] \Rightarrow F \mid (a\#r) \Rightarrow G\ a\ r) = G\ b\ t$$

$$[] \neq a\#t \qquad \text{– distinctness}$$

$$[\![a = [] \rightarrow P; \exists\ x\ t.\ a = x\#t \rightarrow P]\!] \Longrightarrow P \qquad \text{– exhaust}$$

$$[\![P[]; \forall\ a\ t.\ P\ t \rightarrow P(a\#t)]\!] \Longrightarrow Px \qquad \text{– induct}$$

Finally, there is a compiler for primitive and well-founded recursive function definitions. For example, we may define a sorting operation on lists with elements underlying a linear order by:

fun ins :: α::linorder $\Rightarrow \alpha$ list $\Rightarrow \alpha$ list **where**
 ins x (y#ys) = (if x < y then x#y#ys else y#(ins x ys))
|ins x [] = [x]

fun sort :: α::linorder list $\Rightarrow \alpha$ list **where**
 sort (x#xs) = ins x (sort xs)
|sort [] = []

Here, $\alpha :: \text{linorder}$ requires that the type α is a member of the *type class* linorder. Thus, the operation sort works on arbitrary lists of type $(\alpha :: \text{linorder})$ list on which a linear ordering is defined.

[1]Note that we use a different sytanx for the elements of the option type than the one used in the Isabelle library.

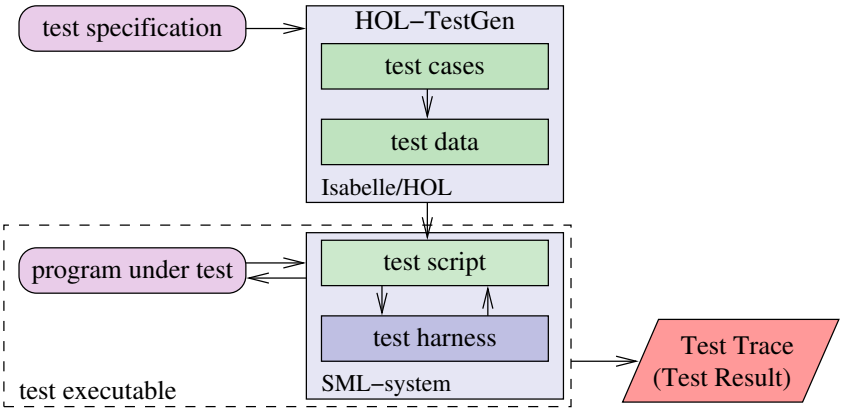

Figure 2.1: Overview of the Standard Workflow of HOL-TestGen

The internal (non-recursive) constant definition for the operations ins and sort is quite involved; however, the logical compiler derives all the equations in the statements above from this definition and makes them available for automated simplification.

Isabelle/HOL also provides a large collection of theories, like for example sets, lists, multisets, orderings, and various arithmetic theories which only contain rules derived from conservative definitions. In particular, Isabelle manages a set of *executable types and operators*, that is types and operators for which a compilation to SML, OCaml or Haskell is possible.

2.3.2 HOL-TestGen

In this section, we present HOL-TestGen [BW12], a model-based testing tool based on the theorem prover Isabelle/HOL. HOL-TestGen is an interactive (i.e. semi-automated) test tool for specification based tests built upon Isabelle/HOL. It allows one to write test specifications in HOL, (semi-) automatically partition the input space to abstract test cases, automatically select concrete test data, and to automatically generate test scripts for testing arbitrary implementations.

2.3.2.1 Workflow

The standard workflow (see Figure 2.1) is divided into five phases:

1. Writing the *test theory*, providing a vocabulary of basic datatypes and definitions.
2. Writing the *test specifications* TS representing the concrete "question" that the system under test is asked.

3. Generation of *test cases* TC, i.e., an abstract representation of a class of test data with constraints.

4. Generation of *test data* TD, i.e., constraint-free instances of TC.

5. Carrying out the *test execution* (*result verification*) phase involving runs of the "real code" of the program under test inside a generated test driver.

The test execution phase will work differently, according to the specific system under test. For many programs, HOL-TestGen provides the means to test them automatically. After the generation of test data, HOL-TestGen can produce a *test script*. The test script contains sequences of abstract inputs and descriptions of possible outputs. The test script, together with a generic *test harness*, performs the test execution on the system under test, this is called the *test execution engine*. It also performs basic test statistics. Often, the generated test execution engine has to be extended by adaptors that map abstract input test data to the concrete input format the system under test uses; the same holds for the concrete output behaviour that must be converted to abstract test output. HOL-TestGen does not provide an automatic compilation of abstract input data to concrete input data of a (reactive) system under test. These adaptors have to be coded by hand.

2.3.2.2 Test Case Generation in HOL-TestGen

The test case generation procedure itself is driven by an exhaustive backward-application of rule of a tableaux calculus combined with normal-form computations eliminating redundancy. Interleaved with this partitioning process (similar to the disjunctive normal form (DNF) of Dick and Faivre [DF93]), test hypothesis rules are generated on the fly and applied to certain subgoals in a backward manner.

In more detail, *test case generation* is a procedure that decomposes a *test specification (TS)*, i.e. a test property over a program under test `PUT`, into a *test theorem* of the form:

$$\frac{TC_1 \cdots TC_n \qquad H_1 \cdots H_n}{TS}$$

where the TC_i are the test cases (or: subdomains of the input/output relation) and where the H_i are the explicit test hypotheses underlying this test. Thus, a test theorem has the following meaning: "If the program under test passes the tests with a witness for all TC_i successfully, and if it satisfies all test hypotheses, it is correct with respect to the TS." A test case generation is called correct if and only if the implication in the test theorem is an equivalence; under this condition, the test theorem bridges the gap between test and verification. The *test data selection* is an automated procedure that

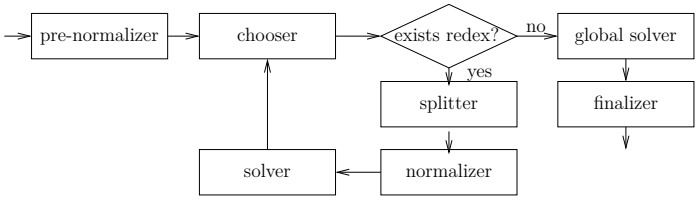

Figure 2.2: A schematic description of the test case generation algorithm.

converts the test cases TC_i of the form:

$$\exists x_1 \cdots x_{ik}.C_{i_1}(x_1, ..., x_{ik}) \wedge \cdots \wedge C_{i_p}(x_1, ..., x_{ik}) \rightarrow P(PUT, x_1, ..., x_{ik})$$

into the formula $P\ (PUT,\ c_1, ...,\ c_{ik})$ where the test data $c_1, ..., c_{ik}$ are ground terms, PUT is a free variable serving as place-holder for the program under test, and P is a closed expression.

The workflow of the test case generation procedure is shown in Figure 2.2. It can be separated into the following phases, which are implemented by largely re-configurable tactics:

Pre-Normalizer: definitions of the test specification may be unfolded.

Chooser: selects (splitting) "redexes", i.e. subterms in the current clause lists on which case-splitting rules will be applied.

Splitter executes case-splitting rules for the selected redexes.

Normalizer: applies a tableaux calculus to split the current list of subgoals into *Horn-clause normal form* (HCNF). By re-ordering the clauses, the calls of the program under test are rearranged such that they occur only in the conclusion, where they must occur at least once; this representation is called *testing normal form* (TNF) if the conclusion is an executable term.

Solver: attempts to eliminate horn-clauses with unsatisfiable constraints.

Finalizer: introduces a uniformity hypothesis for all remaining free variables .

The procedure performs these steps until no more redexes are found.

After the test case generation, the test data generation phase is executed. In this phase, the test theorem is decomposed into the test hypotheses and the test cases, and a constraint resolution phase is invoked on the latter. The constraint solver consists of a combination of standard Isabelle procedures, the SMT solver Z3 [dMB08], and a random solver.

2.3.2.3 An Example

In the following, we present a short example outlining the usage of HOL-TestGen. We use a simple scenario for black box testing of functional programs with a specification based unit test, namely testing sorting algorithms over lists.

Writing the Test Theory We start by specifying a recursive predicate describing sorted lists:

```
fun is_sorted:: (α::linorder) list ⇒ bool where
 is_sorted  []       = True
|is_sorted  (x#xs) = (case xs of []  ⇒ True
                               | y#ys ⇒(x < y) ∨(x = y))
                          ∧ is_sorted xs
```

This predicate takes a list of elements of an arbitrary type on which a linear order is defined as input, and returns True iff that list is sorted.

Writing the test specification We use the defined HOL predicate for describing the test specification, i.e. the properties the implementation should satisfy:

```
test_spec is_sorted(PUT (l::(α list)))
```

where *PUT* is a "placeholder" for the program under test. Informally, this test specification says that the result of applying the program under test to a list should satisfy the *is_sorted* predicate.

Generating test cases Now we can automatically generate *test cases*. Using the default setup, we just apply the test case generation algorithm:

```
test_spec is_sorted(PUT (l::(α list)))
  apply(gen_test_cases PUT)
```

which calculates the test partitioning. At this stage, we can also specify the depth of the test case generation algorithm determining the size used for the regularity hypotheses. Next we bind the test theorem to a particular named *test environment* called the *test theorem*.

```
store_test_thm test_sorting
```

Generating test data Now we want to generate concrete test data, thus all variables in the test cases must be instantiated with concrete values. This involves a random solver which tries to solve the constraints by randomly choosing values.

gen_test_data "test_sorting"

Which might lead to the following set of test data:

is_sorted (PUT [])
is_sorted (PUT [15])
is_sorted (PUT [19, 41])
is_sorted (PUT [31, 65, 41])

Test Execution and Result Verification The test script can be generated by the following command:

gen_test_script "list_script.sml" test_sorting PUT "myList.sort"

Using the test harness we can then build the test executable for the sorting algorithm we want to test. Assuming that algorithm contains an error, the test result might look like the one shown in Table 2.1:

```
Test Results:
=============
Test 0 -     SUCCESS, result: []
Test 1 -     SUCCESS, result: [15]
Test 2 -     SUCCESS, result: [19, 41]
Test 3 - *** FAILURE: post-condition false, result: [41, 65, 31]

Summary:
--------
Number successful tests cases:  3 of 4 (ca. 75%)
Number of warnings:             0 of 4 (ca.  0%)
Number of errors:               0 of 4 (ca.  0%)
Number of failures:             1 of 4 (ca. 25%)
Number of fatal errors:         0 of 4 (ca.  0%)

Overall result: failed
===============
```

Table 2.1: Output of Test Results

2.4 Policy Conformance Testing

Security testing is one of the most difficult testing categories. The reason is that in security, we want to rule out unwanted behaviour. If only one of infinitely many possible

system traces corresponds to a security breach, the system is insecure. Traditional testing approaches, especially random-testing, are unlikely to find that particular system trace.

In this thesis, we aim to address one particular aspect of security testing, namely testing a system's conformance to a given the specification of a security policy. In more detail, we are given a security policy, for example an access control policy, and generate test data from a formal model of that policy. The test data are then used to check if a set of systems implement this policy correctly. We do not need a detailed model of the system under test, only of the policy and the policy-relevant parts of the system.

With this approach, we hope to find a wide range of errors that can harm the security of an application, for example:

- Implementation errors in the application
- Implementation errors in the access control framework of an application
- Ambiguities in the policy
- Misconfigurations of the policy

Why is it so important to test for conformance to security policies? The are several reasons. First of all, security policies are mostly used for critical systems, where non-legitimate access can have severe consequences. Furthermore, they must not be overly restrictive. Consider the access to health care systems, where a wrong reject can have consequences on the health of a patient. As policy configurations tend to be complex, very large, and might change heavily over time, errors and misconfigurations are paramount. Traditional testing and verification approaches do not solve the problem completely. Either they are focused on the implementation correctness of the system, without regarding the concrete configuration the system is supposed to implemented and thus do not cater for wrong configurations and only in a very limited way for wrong implementations of a configuration. Or they are limited to verify only the model of the access control policy, and not the system implementing it. With the work presented in this thesis, we close an important existing gap in the existing techniques.

Traditionally, there is a fine distinction between security testing and functional testing. Our methods however, are somewhat in-between these two worlds. We test for functional aspects of security, for example if a policy has been implemented correctly. As in traditional security testing, one single fault leads to a potentially insecure system and might be exploited by an attacker. However, our testing approach is quite different to for example penetration testing, which it does not replace. For example, we do not test against buffer overflows or attacks by malicious or ill-formed inputs. Testing for compliance to a policy is complementary to other testing techniques.

We believe that a model-based testing approach is the right way to tackle these issues. The test cases we get in this approach are tailored to the concrete security policies the

system is supposed to implement and provide a good coverage of the different branches of the policy. Hereby we are able to test against misconfigurations of the given policy in the system and implementation errors in the policy engine at the same time.

3 A Framework for Policy Specification

This chapter contains the description of a generic framework for policy specification, called the Unified Policy Framework (UPF). At this point, we will introduce the framework, while the next chapter will present usage scenarios of the framework regarding model-based conformance testing. This is followed by two large-scale instantiations of the framework with case studies from the network and from the health-care domain.

The Unified Policy Framework (UPF) is a framework for policy modelling with the primary goal of being used for test case generation. It differentiates itself from other modelling frameworks in several fundamental aspects, that we present below. We start by stating the goals of the framework, then present the foundational ideas, before outlining how to formalise rules and to use them to build up policies. Next we introduce how to combine several policies and in section 3.4 we provide the link to transition systems.

The number of existing policy models is large and many of them are accompanied with tool support in different areas. As many different policy models are conceptually quite similar, much duplicate work has been performed in the past. As an example, consider the case for test case generation from policies, where algorithms consist for specific access control models (for example for privacy [SK08b] or firewall policies [HXCL08]). We strongly believe that many access control models are in fact quite similar, and developing methodologies for generic policy concepts brings a lot of advantages.

Other authors have also recognised the large set of similarities among different access control models. For example, Steve Barker states that "existing access control models are essentially based on the same (small number of) primitive notions, and that these primitives are interpreted in a limited (and limiting) manner" [Bar09]. This is also our understanding, and the existence and the widespread usage of the eXtensible Access Control Markup Language (XACML) is a further evidence for this claim.

Genericity is not just a good thing though, it usually comes with a price. Different policy concepts are being used in different domains, and with a generic framework there is the risk that too little can be achieved or too much complexity is being introduced. Thus, a trade-off must be made between those factors. We apply a similar approach as in PBel: "A policy language should therefore support an abstraction layer that encapsulates domain-specific structure, assumptions or knowledge. Its composition mechanisms should be orthogonal to specifics of application domains, and should so facilitate the applicability of policy patterns across application domains." [BH11]. Applying this ap-

proach, and additionally setting out clearly the limits, i.e. concepts the framework is not supposed to support, helps towards achieving a good trade off. We will present some explicit limitations in section 3.5.

Related to this goal is the need to provide enough flexibility in the model. One observation we made when instantiating the framework was the fact that real-world applications tend to be loosely inspired by abstract frameworks rather than implementing them faithfully. Thus, in order for being able to test a real-world system, a fixed policy model is often of little use.

The primary envisaged usage scenario of the Unified Policy Framework is test case generation. For this reason, the modelled policies should be executable, such that a test driver can be generated executing the tests on a system. Furthermore, we require the policies to be unambiguous: there must not be any conflicting rules.

Security policies are often dynamic: their decision depends on the current state the system is in. For this reason, it does not suffice to model the policy in a way where it provides the access control decision only. Instead, we require a close link between policies on one side and state and state transitions on the other side. This is in contrast to many access control meta-models. Ideally, policies, states, state transitions, and their arbitrary combination can be modelled in an integrated and uniform way.

The framework and the accompanied methodologies are based on the interactive theorem prover Isabelle/HOL. Regarding the policy modelling framework itself, choosing HOL as an implementation language has several advantages, some of which will be presented subsequently. Most of all it allows usage of powerful higher-order constructs, facilitating policy specification. Furthermore, the fact that the framework is integrated into a theorem proving environment, allows specification of powerful policy simplification and transformation techniques, specification of refinement notions, and techniques to reason over properties of a policy. Additionally, it ensures formal correctness of the specifications and methodologies at all levels.

Summing up, the UPF provides the following properties:

- Clear separation of the domain-specific parts from the generic framework
- Small number of core concepts allowing reuse of developed methodologies
- Flexible due to the usage of higher-order constructs
- Tight integration with transition systems
- Formally well-defined
- Providing machine-supported reasoning techniques

3.1 Foundation: Policies as Functions

The aforementioned observations and requirements have led to the fact that the UPF is based on a slightly non-standard representation of the concept of a security policy. Traditionally, a policy is seen as a relation on resources, sets of permissions, etc. In contrast, we emphasise the view that a policy is a policy decision function that grants or denies access to resources, permissions, etc. In other words, instead of modelling the relations of permitted or prohibited requests directly, we model the concrete function that implements the policy decision point in a system. Even though functions and relations are closely related concepts, the choice of a functional representation leads to several practically consequences outlined below.

In more detail, the UPF is based on the following four principles:

1. Functional representation of policies
2. No conflicts are possible
3. Three-valued decision type (allow, deny, undefined)
4. Output type not containing the decision only

Regarding the first principle, the functional view of a policy has the advantage that it is compatible with many different policy models, enabling a uniform modelling framework to be defined. Furthermore, this function is typically a large cascade of nested conditionals, using conditions referring to an internal state and security contexts of the system or a user, making them well suited for automatic test case generation.

A direct consequence of the functional representation leads to the second principle: there are no conflicts possible in a given policy. A request is either allowed, denied, or undefined. This is in contrast to many other policy modelling frameworks, where it is possible to have conflicting rules and which usually require the indication of some conflict resolution strategy. In our framework, the resolution strategy is already implicitly applied when combining the rules to a policy. We do not consider this as a disadvantage: the primary goal of the framework is to test for conformance of a system to a specified policy. In order to make a qualified statement about conformance, the specification needs to provide a well-defined decision statement.

A similar argument might be used against the third principle, namely to additionally provide the possibility of having the output undefined. Again, in practice one would usually like to have a specified policy completely defined, that is a request should either be allowed or denied. The reason why we still need undefinedness is that we describe elementary policies by partial system behaviour, which are then glued together by operators such as function override and functional composition. These elementary policies are often not total, that is they only defined for a part of the input domain. The advantage

of having the concept of undefinedness integrated into a policy modelling framework has also been observed by other authors. For example in [WL93], an undefined value can either mark an oversight or an error or being integrated explicitly to mark a "hole to be filled" later when composing several policies.

The fourth principle, that the output of the policy should not only consist of the decision but allow an arbitrary additional type, has several reasons. First of all, when testing a system for conformance to a policy, we are very often not only interested in the decision alone, but would like to have some further information. This could for example be a transformed packet in the case of a network router performing network address translation, a return message, or a log where some requests need to be stored for further inspection. Furthermore, this allows for tight integration with a model of a stateful system, and is a requirement for being able to specify stateful policies.

The requirement of a functional representation including undefinedness paves the way to model a policy as a partial function from some input data to a decision. This decision is either an allow or a deny together with some other output, or undefined. For achieving the requirement of having the domain-specific parts orthogonal to the composition mechanisms, we use an abstract type for both the input data as well as for the output data.

HOL as a logic of total functions does not provide first-class support for partial functions. Several workarounds have been suggested, for example in [Ola97]. Using their terminology, we employ a "lifting"-approach, with the `option` type from the Isabelle library. This datatype, defined as

datatype α option $= \bot \mid \lfloor \alpha \rfloor$

is used whenever one needs to add a distinguished element to some existing type. In our case, this distinguished element is the undefinedness.

Formally, the concept of a policy is then specified as follows:

datatype α decision $=$ allow α $\mid$ deny α

type_synonym (α,β) policy $= \alpha \rightharpoonup \beta$ decision (infixr $\mapsto$ 0)

where

type_synonym $\alpha \rightharpoonup \beta = (\alpha \Rightarrow (\beta$ option))

Thus, summing up, a policy from input data α to output data β is written as $\alpha \mapsto \beta$, which is a **partial function from** α **to** β **decision**.

Following this definition, the range of a policy may consist of $\lfloor$`accept x`$\rfloor$, of $\lfloor$`deny x`$\rfloor$, as well as $\bot$, modelling the undefinedness of a policy. This is depicted in Figure 3.1.

Any element from the input type α is mapped to either an element in the set of the allowed values, to an element in the set of denied values, or to an undefined value.

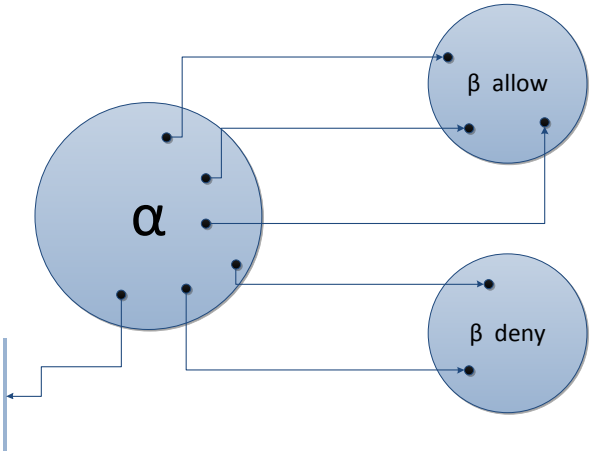

Figure 3.1: Conceptual view of a policy in the UPF

To avoid confusion, we will apply the following terminology to the different parts of the output of a policy in the sequel:

- **decision:** either allow, deny, or undefined
- **outcome:** the additional output element (i.e. the β in β *decision*)
- **output:** the combination of the two (i.e. β *decision* or $undefined$)

The range of a policy thus consists of two fundamental sets: the allow-set *Allow* and the deny-set *Deny*:

definition Allow :: (α decision) set
where Allow = range allow

definition Deny :: (α decision) set
where Deny = range deny

So far, we stayed on the abstract level, not stating what either α or β are in practice. In the following, we outline several simple examples and show how they can possibly instantiate these two type variables (note that there are usually several possibilities). A common case is when we do not have a particular outcome, i.e. the policy specifies for each (defined) input only either an accept or a deny. In this case we would not need any β at all. This can be modelled in the framework by using the *unit* type of the Isabelle library. *unit* is a type that consists of one single element only, which is written (). In Figure 3.1 this would correspond to the set of all allowed values consisting of one element only and equivalently for the set of denied values.

- Consider a policy specifying for each user and resource if he is granted access to

it. The input is a pair consisting of a user and a resource, and as output we need the decision value only, thus using the unit type for the outcome. The type of the policy is: $user \times resource \mapsto unit$.

- Consider again a user accessing a resource, but each user is only granted access to a specific resource once. Thus we need a state σ tracking the granted access requests. The input is now a triple $user \times resource \times \sigma$, and the output is the decision together with a possibly changed state σ. Thus, the type of the policy is: $user \times resource \times \sigma \mapsto \sigma$.

- Consider Administrative Role-based Access Control (ARBAC), where some users can also adjust the role-memberships. One possibility to model this is to have the same type of a policy as in the last example ($user \times resource \times \sigma \mapsto \sigma$), but where the state σ contains the user-role membership relation.

- Consider a stateless packet filter, taking a network packet as input and statically either accepting or rejecting it. Here, α is the packet, and β is $unit$. Thus, the type of the policy is: $packet \mapsto unit$.

- Consider a firewall additionally performing some kind of Network Address Translation (NAT), that is, a packet can be transformed to another one when passing the firewall (for example adjusting the destination IP address). In addition to the stateless packet filter case, the output of the policy now also contains the description of a packet, and consequently the type of the policy is: $packet \mapsto packet$.

Note that we gain a lot of genericity by keeping the input and output types completely open. Many access-control meta-models require their request to be of a specific format (for example a $(subject, object, action)$-triple), unnecessarily restricting the model.

For introduction of the notation used in the sequel, consider a policy of the form $user \times resource \times \sigma \mapsto \sigma$ as outlined in the second example above. Let the policy being modelled be called $MyPolicy$, $Alice$ is a user, $file1$ a resource to be accessed, and σ_1 and σ_2 are states. Evaluating the policy for a specific access is written as:

$$MyPolicy\ (Alice,\ file1,\ \sigma_1)$$

and there are three possible output categories:

- **Accept:** written as $\lfloor \text{accept } \sigma_2 \rfloor$
- **Deny:** written as $\lfloor \text{deny } \sigma_2 \rfloor$
- **Undefined:** written as $\bot$

3.2 Foundational Concepts of the UPF

In this and the next section, we introduce the main concepts and operators of the UPF, that define how rules can be defined and combined to larger policies which can itself be combined with other policies and transition systems. In contrast to the open input and output types, only a relatively small number of concepts and operators is provided, in order to reduce complexity. This ensures that many of the developed methodologies and techniques are applicable throughout the framework, among different kinds of policies. However, it must also be noted that because the UPF is implemented as a shallow embedding in higher-order logics, the provided set of operators is not fixed. If needed in special circumstances, the UPF might easily be extended. However, this should be avoided as long as possible as it leads to increased complexity and would require further proof support for the extended operators.

All the main concepts we introduce in this section are summarised in Table 3.1

3.2.1 Elementary Policies

We start by specifying elementary policies. There are essentially three categories of them, relating to the three possible decision values:

- The empty policy; undefined for all elements: $\emptyset = \lambda \mathrm{x}.\ \bot$
- A policy allowing everything. However, this only defines the decision, not the outcome. Thus, there exist four common variants:
  - with a partial function pf applied to the input to get the outcome: A_p

    definition $\mathrm{A}_p :: (\alpha \rightharpoonup \beta) \Rightarrow (\alpha \mapsto \beta)$
    where $\mathrm{A}_p\ \mathrm{pf} = \lambda \mathrm{x}.\ \mathrm{case\ pf\ x\ }$**of** $\lfloor \mathrm{y} \rfloor \Rightarrow \lfloor \mathrm{allow\ y} \rfloor$
    $\mid \bot \Rightarrow \bot$

  - with a total function f applied to the input to get the outcome: A_f
  - where the outcome is always unit: A_U
  - where the outcome is the identity function on the input: A_I

  The first one is the most general one, and in fact the others are can be defined in terms of that one (for example $A_I = A_p(id \circ \lambda\, x.\lfloor x \rfloor)$). They are still provided as first-class elementary policies as they are widely used.
- A policy denying everything, with the same four variants as for the allowance case.

3.2.2 Domain and Range of a Policy

The notions of a *domain* (dom p ::$[\alpha \mapsto \beta] \Rightarrow \alpha$ set) and a *range* (ran p :: $[\alpha \mapsto \beta] \Rightarrow$ (β decision) set) of a policy are very important when reasoning about policies specified in the UPF. The domain is defined as the set of all input elements, for which the policy is defined, while the range is the set of all outputs (i.e. together with their decision), for which an input value is defined. For the elementary policies above, the domains and ranges are as follows (and equivalently for the corresponding Denial policies):

- $dom\ \emptyset = \{\}$. The domain of the empty policy is empty.
- $ran\ \emptyset = \{\}$. The range of the empty policy is empty.
- $dom\ (A_f f) = UNIV$. The domain of the allow everything policy is the universal set.
- $ran\ (A_f f) = ran\ (\lambda x.\ \lfloor allow\ (f\ x) \rfloor)$. The range depends on the function f. As a special case:
- $ran\ A_U = \{allow\ ()\}$. Thus the range consists of one element only.

Reasoning over the domains of the policies enables specification of powerful policy transformation procedures, as will be presented in section 4.2.

Inspired by the Z notation [Spi92], we introduce the concept of *domain restriction* $\lhd$ and *range restriction* $\rhd$, defined as:

definition _ $\lhd$ _ :: $[\alpha$ set, $\alpha \mapsto \beta] \Rightarrow \alpha \mapsto \beta$
where S $\lhd$ p = λx. if x $\in$ S then p x else $\bot$

definition _ $\rhd$ _ :: $[\alpha \mapsto \beta, \beta$ decision set$] \Rightarrow \alpha \mapsto \beta$
where p $\rhd$ S = λx. if the (p x) $\in$ S then p x else $\bot$

3.2.3 Updates

There are a number of operators that change the result of applying the policy to a particular element. The essential one is the *update*:

p(x$\mapsto$t) = λy. if y = x then $\lfloor$t$\rfloor$ else p y

and its two variants:

p(x+$\mapsto$t) = p(x$\mapsto$allow t)
p(x$-\!\!\mapsto$t) = p(x$\mapsto$deny t)

To make a policy p undefined for a particular element x, we can write:

$p(x{:=}\bot)$

These updates can also be combined. For example to define a policy allowing `x` with outcome `a`, and denying `y` with outcome `b`, and being undefined for everything else, we may write:

$\emptyset(x +\!\!\mapsto a,\ y -\!\!\mapsto b)$

The usual theorems about function updates apply, for example, later updates cancel out former ones as stated in the following theorem:

lemma $p(x{+\!\!\mapsto}a,x{-\!\!\mapsto}b) = p(x{-\!\!\mapsto} b)$

3.2.4 Rules

There are several ways of how to define individual rules in the UPF. One approach, applicable for very simple cases, is to use the update operators just introduced. As an example the following rule allows Alice to read object1:

$\emptyset((\text{Alice},\text{obj1},\text{read}) +\!\!\mapsto ())$

The most often used approach however is to define a rule as a refinement of one of the elementary policies, by using a domain restriction. As an example, the previous example is equivalent to:

$\{(\text{Alice},\text{obj1},\text{read})\} \lhd A_U$

This approach also allows us to define more complex rules. As another example, consider a policy $Integer \mapsto unit$. We would like to have a rule defining acceptance for all values smaller than 10. This can formalised as a domain restriction of the A_U policy to the set of integers smaller than 10:

$\{x.\ x < (10::\text{int})\} \lhd A_U$

3.3 Combining Rules and Policies

Essential to any policy framework are operators allowing one to combine policies (or, equivalently, rules). The most important question to solve when making a combination of policies is how to deal with conflicts. A conflict appears whenever the policies to be combined have a different output for the same input. In the UPF, where the output does not consist of the decision only, there is also the question of how to combine the outcomes. In the way how to deal with conflicts, we see a fundamental difference

Elementary Concepts	dom	The domain of a policy
	ran	The range of a policy
Elementary Policies (equivalent for the Denials)	$\emptyset$	The empty policy
	A_p	Transforming a partial function into a policy allowing everything in the domain
	A_f	Transforming a total function into a policy allowing everything
	A_I	Allow everything, with the identity outcome
	A_U	Allow everything, with the unit outcome
Update Operators	$p(x \mapsto t)$	x is updated with output t
	$p(x+ \mapsto t)$	x is updated with *allow* t
	$p(x- \mapsto t)$	x is updated with *deny* t
	$p(x := \bot)$	x is made undefined
Restrictions	$\lhd$	Domain Restriction
	$\rhd$	Range Restriction

Table 3.1: Basic Operators of the UPF

between the UPF and many other policy frameworks as, for example PBel [Ola97]. In those, seen conceptually, the policies are specified as elements of a list and later on a conflict resolution strategy is applied. In the UPF, due to its functional nature, this is done differently: conflicts can never occur by definition, if conflicting policies are to be combined, the conflicts are resolved immediately, determined by the choice of the specific combination operator. In that respect, the UPF is strictly less general than other frameworks. This is for example relevant when comparing with frameworks integrating break-glass concepts where conflicts are an integral part of the formalism (see for example [BP09, MCMD11]).

There are three categories of combination operators:

- Override operators: used for policies of the same type
- Parallel combination operators: used for the parallel composition of policies of potentially different type
- Sequential combination operators: used for the sequential composition of policies of potentially different type

In the first case, we often talk about combination of *rules*, while in the latter we talk about combination of *policies*. However, this naming is just a matter of taste, conceptually they are the same. The reason is purely pragmatic: We often combine rules of the same type to policies, which we might then combine with potentially differently typed policies.

In Table 3.2, we provide an overview over all available composition operators.

3.3.1 Override Operators

The override operators are the most fundamental ones of the composition operators. They are most often used in the construction of policies out of single rules. They can only be applied to policies having the same input and output type. If the domains of the rules to be combined overlap, one of the two possible outputs is chosen for the resulting policy. In contrast, the two other categories of composition operators evaluate both and combine them according to the specific operator chosen.

The most important operator is the standard left-to-right *override* $\_\oplus\_ :: [\alpha \mapsto \beta, \alpha \mapsto \beta] \Rightarrow \alpha \mapsto \beta$. It allows rules to be combined in a first-fit manner, for example:

$p_1 \oplus \cdots \oplus p_n \oplus D_f$

The policy is applied from left-to-right. The first rule that matches will be applied. In this thesis, this operator is often called the *standard override operator*.

Override Operators	$++$	Last applicable rule is applied
	$\oplus$	First applicable rule is applied.
	$\oplus_A$	First applicable rule is applied, but an allow takes precedence over a deny
	$++_A$	Last applicable rule is applied, but an allow takes precedence over a deny
	$\oplus_D$	First applicable rule is applied, but a deny takes precedence over an allow
	$++_D$	Last applicable rule is applied, but a deny takes precedence over an allow
Parallel Operators	$\otimes_1$	Parallel composition where the decision of the first policy will be applied
	$\otimes_2$	Parallel composition where the decision of the first policy will be applied
	$\otimes_A$	Parallel composition where the decision is allow if any of the two input decisions is an allow
	$\otimes_D$	Parallel composition where the decision is deny if any of the two input decision is a deny
Sequential Operators	$\circ_1$	Sequential composition where the decision of the first policy will be applied
	$\circ_2$	Sequential composition where the decision of the second policy will be applied
	$\circ_A$	Sequential composition where the decision is allow if any of the two input decisions is an allow
	$\circ_D$	Sequential composition where the decision is allow if any of the two input decisions is an allow
Operators for State Transitions	$\otimes_m$	Combines two partial functions in parallel
	$\otimes_S$	Combines two state transitions in parallel
	$\otimes_\nabla$	Parallel composition of two state transitions and a policy
Coercion Operators	∇	Range-modifying operator with two distinct behaviours for both decision values
	o_f	Range-modifying operator with only one behaviour
	o	Domain-modifying operator (functional composition)

Table 3.2: Overview of the operators for composition

The second operator is the symmetric equivalent: the last matching rule will be applied. This operator is written as ++, and the following theorem holds:

x ⊕y = y ++ x

In fact, the ⊕ operator is only defined as syntactic translation of the ++, which is the standard Isabelle override operator:

m1 ++ m2 = λx. case m2 x **of** ⊥⇒m1 x | ⌊y⌋ ⇒⌊y⌋

For both of these overrides, we have two variants. For example, in the case of the operator $\oplus_A$, the first matching rule will be applied, with the exception that an allow in the second rule wins over a deny in the first rule. The case for $++_A$ is equivalent. Following are the definitions of these two operators, where again one of them can be defined as a syntactic translation of the other. The situation for the two operators where a deny wins is dual.

definition _ $++_A$ _ ::[α↦β, α↦β] ⇒α↦β
where m2 $++_A$ m1 =
λx. case m1 x **of**
⌊allow a⌋ ⇒ ⌊allow a⌋
| ⌊deny a⌋ ⇒ (case m2 x **of** ⌊allow b⌋ ⇒ ⌊allow b⌋
| _ ⇒ ⌊deny a⌋)
| ⊥⇒ m2 x

translations
p $\oplus_A$ q == p $++_A$ q

3.3.2 Parallel Composition Operators

Parallel composition of policies is based on the idea that two policies are executed independently in parallel, and then the two outputs of the policies are used to compute the final decision. This behaviour is inspired by the product construction of automata: each policy makes an (independent) step, and the result is a step relation on the Cartesian product of states.

It is important to note that both the input and the output of the policies to be combined can differ; in the resulting policy they are paired. Thus, the type of the parallel composition operators is as follows:

$[\alpha \mapsto \beta, \gamma \mapsto \delta] \Rightarrow (\alpha \times \gamma \mapsto \beta \times \delta)$

When performing parallel composition, the decisions of the policies to be combined are used to compute the decision value of the resulting policy, while the outcomes are simply paired. Thus it must be determined how the decision values shall be combined. There

are actually quite a wide range of different semantic behaviours that could be used in combination (can also observed in the large number of policy combination algorithms of XACML [OAS05]). In our framework we make two important limitations to the parallel composition operators:

- They are only binary. In other words, they only combine two policies in parallel. This rules out modelling *directly* more special combination strategies like a majority vote.
- They are strict. Whenever one of the policies to be combined is undefined for an input, so will be the resulting policy. Note that many XACML policy combinators do not have this restriction.

Having these two limitations reduces the possibilities of combination of the decision values to four. Thus there are four parallel composition operators:

- Take the decision of the first policy (called $\otimes_1$)
- Take the decision of the second policy (called $\otimes_2$)
- If allowed by any policy, return allow (called $\otimes_{\vee A}$)
- If denied by any policy, return deny (called $\otimes_{\vee D}$)

The question might arise what the purpose of the first two possibilities to combine a decision actually is when combining total policies. It seems as the second (resp. the first) policy will never be taken into account. However, this only holds for the decision itself, the outcome is still considered. We will later encounter examples where these operators are useful.

As an example of how such a parallel combinator is defined, consider $\otimes_{\vee D}$. The other cases proceed analogously.

```
definition _⊗∨D_ :: (α ↦β) ⇒(γ ↦δ) ⇒(α ×γ↦β×δ)
where p1 ⊗∨D p2 =
λ(x,y). case p1 x of
      ⌊allow d1⌋ ⇒ (case p2 y of
                      ⌊allow d2⌋ ⇒ ⌊allow (d1,d2)⌋
                    | ⌊deny d2⌋ ⇒ ⌊deny (d1, d2)⌋
                    | ⊥⇒ ⊥)
    | ⌊deny d1⌋ ⇒ (case p2 y of
                      ⌊allow d2⌋ ⇒ ⌊deny (d1, d2)⌋
                    | ⌊deny d2⌋ ⇒ ⌊deny (d1, d2)⌋
                    | ⊥⇒ ⊥)
    | ⊥⇒ ⊥
```

Often, it is afterwards necessary to adapt the input type or the outcome type of a policy to a more refined context. This boils down to variants of functional composition for functions that do not have the format of a policy. With the standard function

composition $\circ$, it is possible to change the input domain of a policy $p :: (\beta \mapsto \gamma)$ to $p \circ f :: (\alpha \mapsto \gamma)$, provided that f is a coercion function of type $\alpha \Rightarrow \beta$.

Range-modifying operators decide which part of the outcome is chosen; this might be the first, the second or, an arbitrary combination of them. A different strategy may be employed for each decision value. To achieve this, the ∇ operator can be used, defined by:

$$
\begin{aligned}
(\mathrm{f},\mathrm{g})\ \nabla\ \mathrm{p} = \lambda \mathrm{x}.\ \mathrm{case}\ \mathrm{p}\ \mathrm{x}\ \mathbf{of}\\
\quad \lfloor \mathrm{allow}\ \mathrm{x} \rfloor \Rightarrow \lfloor \mathrm{allow}\ (\mathrm{f}\ \mathrm{x}) \rfloor\\
\mid \lfloor \mathrm{deny}\ \mathrm{x} \rfloor \Rightarrow \lfloor \mathrm{deny}\ (\mathrm{g}\ \mathrm{x}) \rfloor\\
\mid \bot \Rightarrow \bot
\end{aligned}
$$

The operator o_f is a simplification of this operator and used whenever f and g are equal:

$$\mathrm{f}\ \mathrm{o}_f\ \mathrm{p} = (\mathrm{f},\mathrm{f})\ \nabla \mathrm{p}$$

To summarise, the following needs to be decided when combining two policies in parallel:

- how to combine the decisions: this is done by the choice of the operator taken
- (possible) adaption of the resulting input type (applying functional composition o)
- (possible) adaption of the resulting outcome type (applying ∇ or o_f)

As an example, consider the case where we would like to combine the following two policies:

- A policy $P1$ of the form $user \times permission \mapsto outputMSG$, taking a user and a permission as input, and displaying an output message together with the decision, and
- a policy $P2$ of the form $user \times permission \times \sigma \mapsto unit$, which additionally takes a state σ as input, but only has the decision as output.

We would like to combine them to a policy $P3$ of the form:

$$user \times permission \times \sigma \mapsto OutputMsg$$

The decision should be undefined if it is undefined by any of the two policies, and else be denied if denied by any of the two policies.

This can be formalised as follows:

$$P3 \ = \ (\lambda(x,y).\ x)\ o_f\ (P1 \otimes_{\vee D} P2)\ o\ (\lambda(a,b,c).\ ((a,b),(a,b,c)))$$

If we would just combine the two policies using $\otimes_{\vee D}$, we would end up with a policy of

the type:

$$((user \times permission) \times (user \times permission \times \sigma)) \mapsto (OutputMsg \times unit)$$

the first coercion function above adapts the range by throwing away the unit type, the second one adapts the domain by identifying the two occurrences of user and permission.

3.3.3 Sequential Composition Operators

Sequential composition is based on the idea that two policies are to be combined by applying the second policy to the output of the first one. In that respect, composition is similar to functional composition. The types of the two policies cannot be arbitrary: the outcome of the first policy must be of the same type as the input to the second one. We use the same order of the operators as for the standard Isabelle composition operator, where a ∘ b stands for a applied to the result of b, defined as λ x. a (b x).

Thus, the sequential policy combination operators are all of the type:

$(\beta \mapsto \gamma) \Rightarrow (\alpha \mapsto \beta) \Rightarrow (\alpha \mapsto \gamma)$

The sequential operators have the same two properties as the parallel ones:

- They are strict regarding undefinedness
- They are only binary (but of course sequences of sequential compositions are admissible)

As an example where sequential composition can be applied, consider a network policy where several firewalls and routers may perform filtering and address translation techniques to packets going through: a firewall applies the policy to packets as they were returned by the previous one.

Similar to the parallel composition, there are several possibilities of how to combine the decision values. We distinguish the following four cases, leading to four sequential composition operators. Again, in all cases, the resulting decision will be undefined if it is undefined by either of the two policies to be combined.

- Take the decision of the first policy (called $\circ_1$)
- Take the decision of the second policy (called $\circ_2$)
- Allow if allowed by any policy (called $\circ_{\vee A}$)
- Deny if denied by any policy (called $\circ_{\vee D}$)

Isabelle provides a composition operator on partial functions, defined as:

definition _ o_m _ :: $(\beta \rightharpoonup \gamma) \Rightarrow (\alpha \rightharpoonup \beta) \Rightarrow (\alpha \rightharpoonup \gamma)$
where f o_m g = λk. case g k **of** $\bot \Rightarrow \bot$ | ⌊v⌋ ⇒ f v

Since policies in our framework are partial functions, we can reuse this composition operator. However, this will result in nestings of decisions, which must be flattened. There are only four flattening possibilities as undefinedness is always strict, leading to the four operators introduced above. For example, the flattening for the case where the resulting decision is an allow whenever any of the input decisions is an allow, can be defined as follows:

fun flat_orA :: (α decision) decision $\Rightarrow$ α decision
where flat_orA(allow(allow y)) = allow y
|flat_orA(allow(deny y)) = allow y
|flat_orA(deny(allow y)) = allow y
|flat_orA(deny(deny y)) = deny y

Next we require a lifting operator that ensures that decisions are transferred:

fun lift :: ($\alpha \mapsto \beta$) $\Rightarrow$ (α decision $\mapsto \beta$decision)
where lift f (deny s) = (case f s **of** $\lfloor$y$\rfloor$ $\Rightarrow$ $\lfloor$deny y$\rfloor$
| $\bot \Rightarrow \bot$)
| lift f (allow s) = (case f s **of** $\lfloor$y$\rfloor$ $\Rightarrow$ $\lfloor$allow y$\rfloor$
| $\bot \Rightarrow \bot$)

Finally, we define the sequential policy composition operators:

definition _ $\circ_{\vee A}$ _ :: [$\beta \mapsto \gamma$, $\alpha \mapsto \beta$] $\Rightarrow \alpha \mapsto \gamma$
where p2 $\circ_{\vee A}$ p1 = (Option.map flat_orA) o (lift p2 o_m p1)

That is, in detail, first the two policies, with the second one lifted, are composed using the standard partial function composition operator, leading to a partial function with nested decisions ($\alpha \rightharpoonup$ (β decision decision)). The resulting function is combined using the standard function composition operator with the respective flattening function which was transformed to a partial function.

The other three operators are defined analogously, by using one of the other flattenings.

As in the parallel case, we may apply coercion functions to the combined policies. However, more common in sequential composition is the case that the input or output type of any of the constituent policies needs to be adapted before combining.

3.4 Transition Policies

The intended usage of the UPF is to generate test cases for dynamic systems enforcing some kind of security policy. For this reason, modelling system behaviour needs to be tightly integrated into the framework. In this section, we present how policies and system behaviour can be modelled uniformly leading to what we call a *transition policy*.

The general strategy is to model the system behaviour and the security (access control) behaviour independently and combine them in the end to a combined secured system model.

How policies (often stateless) can be modelled was shown in the previous sections. The system behaviour itself is modelled using *state transitions*. The only thing the UPF requires, is that they are modelled as partial functions. The particular input and output types are application specific. Modelling state transitions as partial functions allows a tight integration with policies. In fact, a state transition could even be called a *policy without a decision value*. Generally, such a state transition has one of the following types:

$$\iota \times \sigma \rightharpoonup \sigma \qquad \iota \times \sigma \rightharpoonup o \times \sigma$$

where ι is an input type (for example a user and an operation to be executed), o some output type, and σ a state.

The exact way of how these state transitions are defined is application specific. However, usually a similar modular approach as used in the policy modelling is also used for modelling state transitions. In other words, we model them in small parts and combine them in parallel to a larger state transition. To combine state transitions in parallel, we provide a parallel composition operator for partial functions, inspired by the parallel composition operators for policies:

```
definition _ ⊗m _ :: (α ⇀β) ⇒(δ ⇀ γ) ⇒ (α ×δ  ⇀ β ×γ)
where p1 ⊗m p2 =
 λ (x,y). case p1 x of ⌊d1⌋ ⇒ (case p2 y of ⌊d2⌋ ⇒ ⌊(d1,d2)⌋
                                           | ⊥ ⇒ ⊥)
                       | ⊥⇒ ⊥
```

Similar coercion functions can be applied as before when combining policies. For the common special case where the state transitions to be combined have the same input but a different state, we provide a distinct operator as syntactic sugar:

```
definition _ ⊗S _ :: (i ×σ⇀ σ) ⇒ (i ×σ' ⇀ σ') ⇒ (i ×σ ×σ' ⇀ σ ×σ')
where p1 ⊗S p2 = (p1 ⊗M p2) o (λ (a,b,c). ((a,b),a,c))
```

Having the necessary individual parts defined, the standard construction methodology for a combined system is the following: the state transitions are defined in two distinct parts. The first one for the transitions which shall be applied whenever the policy grants the respective request, the second one whenever it denies it. In a sense, we construct two behavioural automata of the system, first one for the success case, and second one for the exceptional case. These two automata are then combined with the decision policy, leading to a combined behavioural description of the system implementing a policy. This concept is called a ***transition policy***, as they combine the policy and the transition behaviour of the system into one single definition. Such a policy has the following type:

$(\iota \times \sigma) \mapsto (o \times \sigma)$

where ι is some kind of an input, o some (usually observable) output, and σ the (usually internal) state. If no special output is required, it may just be set to unit. Note that at this stage, we require the state of the input and of the output to be of the same type.

The three parts (the allowance state transition A, the denial state transition D, the policy p) can be combined using the following operator:

```
definition _ ⊗▽ _ :: [(α ⇁γ) ×(α ⇁γ), δ ↦β] ⇒ (δ ×α) ↦(β ×γ)
where (A,D) ⊗▽ p =
 λx. case p (fst x) of
     ⌊allow y⌋ ⇒ (case (A (snd x)) of ⊥⇒ ⊥
                                    | ⌊z⌋ ⇒ ⌊allow (y,z)⌋)
   | ⌊deny y⌋ ⇒  (case (D (snd x)) of ⊥⇒ ⊥
                                    | ⌊z⌋ ⇒ ⌊deny (y,z)⌋)
   | ⊥⇒ ⊥
```

This combinator is a special case of both a parallel composition and a range splitting operator. First, the policy is evaluated. Depending on the decision, one of the two state transitions is taken and the result is paired. Again, this combinator is strict regarding undefinedness and the usual coercion operator may be used afterwards whenever required.

An alternative, equivalent, characterisation of this operator is the following:

```
(A, D) ⊗▽ p =
(((p ▷ Allow) ⊗∨A (A_p A)) ⊕
 ((p ▷ Deny) ⊗∨A (D_p D)))
```

In this definition, all elements for which the policy returns an allow are combined with the policy created from state transition A, and all elements for which the policy returns a deny, are combined with the policy created by state transition D. The two policies are then combined with the standard override operator.

An interesting point has to be mentioned at this point: the state σ in question may itself contain a policy which might get transformed in the state transition. When combining this with a static policy to a transition policy, we actually get a *second-order policy*: the policy specifies how policies can be transformed. As an example of this, we will present the specification of a stateful FTP policy in section 6.2.

To conclude, a transition policy is defined as having the format:

```
(ι ×σ) ↦(o ×σ)
```

In section 4.3, we will present how such a transition policy can be used for model-based testing of a system to a stateful policy.

3.5 Limitations

The term security policy is very broad, and even though the goal of the Unified Policy Framework is to make it usable for a wide range of different kinds of policies, there are many for which it is not suitable. In section 2.1 we already outlined which kinds of security policies the framework should be able to deal with. Here we would like to emphasise on several rather technical aspects of policy concepts that we either cannot model at all, or for which we do not currently provide direct support. Most of these limitations are here on purpose, reducing unwanted complexity.

- No ambiguities. Sometimes it might be desirable to specify ambiguous (sub-) policies, i.e. policies with conflicting outputs. This is not possible in the UPF, as we employ a functional representation - leading to exactly one output for each input. In [MCMD11], the authors claim that conflicting values can be useful when specifying break-glass policies to show "that there is evidence to support both a permission and a denial". The UPF provides no support for that.

- The former can be circumvented in certain cases by specifying all possible outcomes as members of a set. However, we can only specify one decision for each input. This clearly limits the possibilities of modelling non-determinism. As an example, consider a packet going through a firewall undergoing an address translation to a pool of addresses. That is, there is a set of possible outcomes which are admissible and would lead to successful tests. What is not possible however, is to specify directly a rule that would say *allow this packet if the address was rewritten to A, and reject it if rewritten to B.*

- Limited form of representing temporal policies. There is no first-class support for standard temporal operators. In section 4.3 we show how some temporal properties can be expressed for usage in test specifications by providing an according test specification automaton. However, temporal concepts are not first-class citizens of the UPF.

- Some authors claim that "a natural and straightforward policy is the one stating that the most specific authorisation should be the one that prevails" [dVSJ05]. However, we don't believe such operators would be very useful: it should be made explicit which combination strategy is used. Furthermore, in most cases, the term "most specific" is not clear.

- There are only two possible decision values plus undefined. This number is fixed. There is, for example, no specific *error* decision, as postulated for example by [TK06].

- No obligations (see for example [GF05]). There is no direct support for modelling obligations, even though some parts of this concept can be modelled using stateful transition policies.

3.6 Policy Analysis

An important property of every policy framework is its ability to analyse policies and to reason about them. In the case of the UPF, this is even more important as it is primarily used for theorem-prover based testing where the availability of powerful deduction mechanisms greatly influences test case generation. In this section, we focus on several generic policy analysis observations not directly related to testing.

Having the UPF integrated into the theorem prover Isabelle/HOL provides one directly with powerful techniques for reasoning about policies, as the powerful proving techniques provided by the Isabelle framework can directly be applied to UPF policies.

3.6.1 Expressing Policy Properties

A couple of standard properties of policies can directly be specified. As an example, we formalise those properties mentioned by the authors in [BH11]. Some of them have to be slightly adapted to fit into the UPF concepts, and there is one for which there is no direct correspondence (*policy p is conflict-free*).

- *Policy p has no gaps.* In our terminology this reads as the policy is defined for all inputs:

  gap_free p = (dom p = UNIV)

- *Policy p is more defined than policy q*:

  more_defined p q = (dom q ⊂dom p)

- *Policy p is more permissive (rejective) than policy q.* That is, all inputs x for which the decision of policy *p* is allow (deny), are also allowed (denied) by policy *q*:

  more_permissive p q =
  ∀ x. (case q x **of** ⌊allow y⌋ ⇒ (∃ z. (p x = ⌊allow z⌋))
  | ⌊deny (y)⌋ ⇒ True
  | ⊥⇒ True)

- *Policies p and q are equivalent over domain d*:

  eq_dom p q d = (∀x∈d. p x = q x)

- *Policies p and q have no conflicting decisions*:

  no_conflicts p q = dom p = (dom q ∧(∀x ∈(dom p).
  (case p x **of** ⌊allow y⌋ ⇒ (∃z. q x = ⌊allow z⌋)
  | ⌊deny y⌋ ⇒ (∃z. q x = ⌊deny z⌋))))

Note that for the last example, the outcome of the two policies does not necessarily have to be the same (not even of the same type). We merely require their decision values to be equivalent.

Extended properties over these properties can usually be proved with little effort in Isabelle. For example, the following theorem states that the policy transforming any given function into an allowance policy is more permissive than any policy:

lemma more_permissive (A_f f) p

As another example, if p and q have the same domains, and both are more permissive and more rejective than the other one, there must be no conflict:

lemma policy_eq:
  assumes more_permissive p q
  assumes more_permissive q p
  assumes more_rejective q p
  assumes more_rejective p q
  assumes dom p = dom q
  shows no_conflicts p q

An interesting property of the UPF is the fact that most of these properties can be proved quite easily, especially as long as we are not dealing with transition policies. This is due to the fact that policies are essentially partial functions for which Isabelle provides powerful built-in deduction mechanisms.

3.6.2 Algebraic Properties of the Operators

The composition operators mentioned earlier share a couple of algebraic properties. Many of them are of importance when analysing composite policies, and are equally important when transforming and testing composite policies. Here we would like to mention a couple of example properties of the UPF composition operators, without aiming for completeness.

An essential property for most parallel and sequential operators is that the empty policy $\emptyset$ cancels out any policy. This is due to the fact that undefinedness is strict in most cases. For example:

- $p \otimes_{\vee D} \emptyset = \emptyset \otimes_{\vee D} p = \emptyset$
- $p \circ_{\vee A} \emptyset = \emptyset \circ_{\vee A} p = \emptyset$

In contrast, for the override operators, the empty policy is neutral:

- $\emptyset \oplus p = p \oplus \emptyset = p$

Almost all operators are *associative*, as for example the override operators:

- p1 $\oplus_A$ (p2 $\oplus_A$ p3) = (p1 $\oplus_A$ p2) $\oplus_A$ p3

Commutativity does not generally hold. While most parallel combination operators are pseudo-commutative (i.e. modulo some type coercions), the override operators are only commutative if their domains are disjoint:

- p $\oplus$q = q $\oplus$p if dom p $\cap$ dom q = {}
- p $\otimes_{\vee D}$ q = ((λ(x,y). (y,x)) o_f (q $\otimes_{\vee D}$ p)) o (λ(a,b).(b,a))

The sequential composition operators are clearly not commutative.

Distributivity is a very important concept regarding the transformation of combined policies, as explained in further detail in section 4.2. There is a wide range of distributivity properties that hold over the UPF operators. One example is distributivity of the standard override operator over the parallel composition operators:

(P1 $\oplus$P2) $\otimes_{\vee D}$ Q = (P1 $\otimes_{\vee D}$ Q) $\oplus$(P2 $\otimes_{\vee D}$ Q)

3.7 Summary

In this chapter, we have introduced the Unified Policy Framework (UPF), a generic framework for policy specification. We have introduced the general concepts, its composition techniques, the link to transition systems, and some policy analysis methodologies. The Unified Policy Framework differentiates itself from other policy formalisms by its view of a policy as a policy decision function. This choice has a number of consequences helpful in providing a generic framework for model-based testing based on composition principles integrating a model of stateful systems. The framework has a formally well-defined foundation, and due to its integration into a theorem-proving environment it provides powerful machine-supported reasoning techniques.

4 Model-based Policy Conformance Testing

In this chapter we present how the Unified Policy Framework (UPF) can be applied to model-based conformance testing of systems having to implement a policy. In the first section, we will introduce unit testing, the next one will be devoted to the concept of policy transformation procedures: a technique for improving the test case generation process drastically. The third section will present sequence testing and in the final part of this chapter, we will present an overview of how the tests can be executed on real systems.

4.1 Unit Testing

4.1.1 Definition

Testing of systems implementing a policy defined in the Unified Policy Framework can be put into the category of *specification-based testing*, where the test suite is produced according to a specification. The test data can broadly be categorised into two categories: unit tests and sequence tests. The term *unit test* is overloaded in the testing literature, but the standard definition is:

Definition 4.1.1 ***Unit Testing*** *is the testing of individual hardware or software units or groups of related units. [GKM+91]*

In more detail, it usually denotes a white-box testing approach where the smallest unit of code is tested by relatively small tests individually. In that respect, it differentiates itself from other, larger-scale testing approaches like for example module testing.

In our framework, this meaning of the term unit testing is not suitable, as we are using a black-box testing approach. Our definition of the term is loosely based on [ZHM97], and can be phrased as:

Definition 4.1.2 *Testing of a (policy-implementing) system where each test case consists of one independent access request (or operation) with some observable output.*

That is, each test case consists of exactly one access to the system with the observable output. The ordering of the test cases is irrelevant. In sequence testing on the other hand, as introduced later, each test case consists of an ordered list of accesses and the expected output of each of them. Note that our understanding of the term unit testing is actually not limited to policy testing alone, and consequently the definition is often generalised.

4.1.2 General Approach

We will introduce a general approach to unit testing in the UPF with the help of a very small pseudo policy, on which we can outline what happens and what needs to be done in the general case.

The policies take as input a `(subject,object,access)`-triple, where for presentation purposes both `subject` and `object` are modelled as integers and `access` is a datatype consisting of the three elements `read`, `write`, and `delete`. The available subjects and objects are not constrained. The outcome of the policy is the `unit` type.

type_synonym Object = int
type_synonym Subject = int
datatype Access = read | write | delete

type_synonym policy = (Subject ×Object ×Access) ↦unit

The policy consists of a number of rules combined using the standard override operator. Each rule allows one specific kind of access request and there is a default deny rule in the end. As an example, a policy consisting of only one rule besides the default one is defined as:

definition rule1
where rule1 = {x. subject x = 1 ∧object x = 1 ∧access x = write} ◁ A_U

definition policy1 :: policy
where policy1 = rule1 ⊕D_U

Before generating the test cases, we need to state the *test specification*. The test specification - not to be confused by the term *test case specification* - is a statement about the purpose of the test. It is a logical statement that holds if:

- all generated tests produced out of it pass successfully
- all test hypotheses produced out of it hold (recall section 2.2)

In our framework, the test specification is stated as a theorem and consequently standard proving steps can be performed on it. The general format for a unit test specification

for UPF policies is as follows:

$\bigwedge$x. P x $\Longrightarrow$SUT x = Policy x

where `SUT` is a placeholder for the *System Under Test*. The meaning of the test specification is as follows: for all possible inputs `x` satisfying a test predicate `P`, the system under test displays the same behaviour as the modelled policy `Policy`. The predicate `P`, which can be quite complex, serves to constrain the possible input space. As an example it is often used to describe various wellformedness-conditions (for example in our example we might require all input values to be positive integers only), or to limit the domain of interest (for example only testing for users in a given set).

In our example, we do not impose any restrictions and thus simply state the test specification as:

$\bigwedge$ x. SUT x = policy1 x

Next, the test cases can be generated. The sketch of the algorithm has been presented in section 2.2, here we focus on its effect on a UPF policy. The algorithm can be executed by issuing an Isabelle tactic as follows:

apply (gen_test_cases SUT)

with the option to include theorems to be added to the simplifier during the test case generation. Definitional statements about the modelled policy and rules are by default not included in the simplifier. Thus the algorithm fails to produce a result at this stage, as a priori nothing is known about the term `policy1`. For this reason, we either need to include the corresponding lemmas for use in the algorithm, or apply an interactive proof-step before application of the algorithm.

In order to see better what happens internally, we opt here for the latter choice. (Almost) completely unfolding and simplifying the aforementioned policy will lead to the following subgoal:

(subject x = 1 $\wedge$object x = 1 $\wedge$access x = write $\longrightarrow$SUT x = $\lfloor$allow ()$\rfloor$) $\wedge$
((object x = 1 $\longrightarrow$subject x = 1 $\longrightarrow$access x $\neq$write) $\longrightarrow$ SUT x = $\lfloor$deny ()$\rfloor$)

which is equivalent to:

(subject x = 1 $\wedge$object x = 1 $\wedge$access x = write $\longrightarrow$ SUT x = $\lfloor$allow ()$\rfloor$) $\wedge$
(subject x $\neq$1 $\longrightarrow$SUT x = $\lfloor$deny ()$\rfloor$) $\wedge$
(object x $\neq$1 $\longrightarrow$SUT x = $\lfloor$deny ()$\rfloor$) $\wedge$
(access x $\neq$write $\longrightarrow$SUT x = $\lfloor$deny ()$\rfloor$)

Note that this representation is inherent to the way policies are modelled in the UPF, and independent from any test case generation algorithm. In almost all cases, any UPF policy will be reduced to such a format, which, although represented slightly differently,

is conceptually similar to a large if-then-else cascade. This applies also when the policy is constructed via other composition operators and changes only in the sequence testing case presented later in this chapter. Applying the algorithm will transform this cascade into a set of test cases, proof obligations, and test hypotheses - while some of the last two might already have been removed as detected to be tautologies by the theorem proving engine. Essentially, for each possibility a test case is generated. In this example, the result will be four test cases, corresponding to the four cases of the example above:

- `SUT (1, 1, write) = ⌊allow ()⌋`
- `SUT (x,y,z) = ⌊deny ()⌋`, with `x` $\neq$ `1`
- `SUT (x,y,z) = ⌊deny ()⌋`, with `y` $\neq$ `1`
- `SUT (x,y,z) = ⌊deny ()⌋`, with `z` $\neq$ `write`

Next, we might possibly first simplify the constraints of the test cases, or directly store the test theorem in the theorem database of Isabelle. This theorem states that providing all the test data created from the test cases by instantiations satisfying the constraints of the test cases pass, and all test hypotheses hold, the test specification is true. As usual, this theorem can be used in other proofs if required.

Regarding testing, the next step is *test data generation*, where, for each test case, the abstract variables are instantiated with ground values that satisfy the proof obligations. As the detailed workings of this mechanism is irrelevant for the content of this thesis, we refrain from a detailed description. Essentially it is a mixture of constraint solving techniques and a random solver.

In our example, test data generation is trivial, and we might end up with a set of test data as the following (using the same ordering as above):

- `SUT (1, 1, write) = ⌊allow ()⌋`
- `SUT (5, 9, read) = ⌊deny ()⌋`
- `SUT (4, 7, write) = ⌊deny ()⌋`
- `SUT (4, 3, delete) = ⌊deny ()⌋`

4.1.3 Coverage

The quality of any testing method is determined by its ability to find failures effectively. The notion of achieved coverage is often used as a metric for an assessment of this effectiveness. A wide range of different coverage criteria are available in the literature (for example [ZHM97, AOH03]). Since we are in a black-box testing scenario, a *specification-based*, or sometimes also called *model-based* coverage criterion is of most use here. Such coverage criteria are traditionally quite distinct from implementation-based coverage criteria, as for example code coverage. A notable example, trying to integrate both concepts, is [KGTB07]. However, we will concentrate on specification-based criteria in the sequel. Unfortunately, most specification-based coverage criteria are restricted to

a particular formalism. In many cases we may link the concepts to the ones of our framework.

In general, test case generation transforms the input space of the model of the specification into subspaces, respectively subsets[1]. In this section, we investigate the effects of this transformation, and to what coverage this will lead to. Please note that we focus here on the coverage regarding testing of the policies as defined in the UPF, rather than on coverage criterion in relation to the generic test generation algorithm of HOL-TestGen.

As explored in the last section, policies of the UPF are first transformed into a format which is essentially an if-then-else cascade. This allows us to provide two kinds of coverage indications: one on the level of rules of the policy, the other on the level of logical expressions.

Regarding the first category, a number of coverage criteria for specific access control policies have been suggested. As an example, [MX07b] introduces several criteria for XACML policies, while the same authors also provide related criteria for firewall policies in [HXCL08]. Given the non-standard view of a policy in the UPF and its genericity, they cannot be transferred directly. In general however, they usually decompose a policy into a set of rules, predicates, and clauses, and define straightforward coverage criteria upon them. As such, the closest concept which describes the coverage achieved in unit testing UPF policies is the ***Clause Coverage Criterion***, requiring each clause in each rule to be evaluated to true and false at least once. For other policy concepts, this criterion is also called ***predicate coverage***.

As mentioned before, a policy is internally represented as being a large if-then-else cascade. This gives rise to coverage criteria stemming from the domain of coverage criteria for logical expressions, as introduced for example in [AOH03]. Exploring the effects of the test case generation on the example policy of the last section already provides a good indication of the achieved coverage: all possible paths of the specification have been considered, and we get a 100% ***Multiple Condition Coverage*** on the model in that respect.

So far, we have only talked about coverage regarding the specification. However, one of the most common quality metrics for testing is code coverage. In the specific setting of our framework, which is inherently black-box, we can say the following concerning code coverage:

- It is not possible to quantify in the general framework the achieved coverage, as the tests are generated completely independent from the code.
- In general, the models used for model-based testing are more abstract than the code and simplifications thereof. For this reason, code coverage of model-based

[1]Note that this is not necessarily a partitioning as the subsets may overlap, leading to a not necessarily minimal test set.

testing approaches is mostly not very high, with some practitioners estimating an average of about 40-60% coverage (see [Ejs11]).

- If the code to be considered is very close to the specification, we will achieve a very high code coverage in our framework, due to the coverage considerations presented in this chapter. However, often the policies are integrated into the application and the decisions are being made as part of the usual application workflow. As the model which is used for test generation is only specifying the policy, large parts of the code will not be executed when executing the tests.

For these reasons, we believe specification-based coverage metrics to be a better quality metrics for these kinds of tests. For many systems the framework can be applied to (for example the applications of the NPfIT as presented in chapter 5), the implementation of the policy is just one part of the system behaviour. It is expected that the code of the system is already tested by other means. The purpose of the framework is to test for conformance to a specific policy, that is, if it is configured correctly and if the system implements the configuration correctly.

4.1.4 Effect of the Coverage on Scalability

In order to be able to better approximate the achieved coverage regarding the concrete modelling concepts of the UPF, and to assess what the consequences on concrete policy models are, we will present here some results of what happens when gradually adding more complexity to an elementary policy. We will focus here on the mere coverage, i.e. the kind and the number of the generated test cases on an abstract policy model. More experimental results including time measurements, are provided in the chapter about the firewall case study.

It must be noted that the direct effect of the coverage to the general policy model is not trivial to assess, as we apply an interactive approach to testing. In other words, the specification can explicitly be simplified, transformed, or rewritten before application of the test case generation algorithm. Furthermore, some aspects of the policy model can remain closed, while others are completely unfolded. All these factors determine how the policy model is presented to the test case generation algorithm.

We start the investigation by using the policy introduced in the previous section, but where the `Access` is also modelled as integer in order to have a more uniform setting. The starting policy consists of one rule besides the default one. This rule is a statement about three distinct elements of the input to the policy. The achieved coverage is a split of the input space into four categories: one where all the conditions for the single rule to be applied hold, and three for which always one of these conditions is not true. If the rule would only specify one parameter of the input (for example the subject), only two test cases would be generated: one where this parameter is set to the desired value, one

where it is not.

Having one rule will lead to four test cases. Adding one additional rule of the same format, but restricting all elements to different values, will generate eleven. Two of them are the only two allowed possible input cases. The remaining nine test cases are all the combinations (3^2) of the three elements mentioned in each rule not set to one of the specified values. This behaviour carries over when adding additional rules. Consequently, the formula for the number of test cases for a policy of this kind with n rules is:

$$f(n) = 3^n + n \tag{4.1}$$

The next effect we would like to investigate is what happens when the input space is enlarged. In the previous example, the input type was a three-tuple. We now investigate the effect when adding one additional element. Apart from that, the policies remain the same. The resulting numbers are as follows: we start with one additional test case, and afterwards the number approximately quadruples with each additional rule. We may thus generalise above formula to:

$$g(n, k) = k^n + n \tag{4.2}$$

where k is the number of variables in the input space.

Note that in practice not all rules of a policy are so uniform. Variations of the policy rules can lead to either less or more generated test cases. Most notably, the following two reasons can influence the number of test cases in addition to the number given by the formula:

- Often the rules are not completely disjoint on all input variables, so some cases and branches concur (imagine the trivial case where each additional rule is the same as the previous one: the number of test cases will be independent of the number of rules).
- Our input variables are single integers only. For more complex types, the algorithm might perform additional splittings, for example consider the case for lists.

To summarise, when constructing a policy by a sequence of rules composed by an override operator, there is an exponential blow-up with each additional rule, due to the fact that each part of every rule is "combined" with each part of every other rule. The base of the exponentiation and the constant factor depend on the size of the input type (i.e only those part matter for which there is at least one rule using it).

Next, we outline what happens when composing two policies in *parallel*. Not surprisingly, the number of generated test cases is the product of the number of test cases of the

individual subpolicies. As an experiment, we made a parallel composition of the two policies used above by always combining two policies with the same number of rules. The rules of both policies are exactly the same, just with the exception that those of the second one have one additional input variable. Their outcome is unified, but the domain is left intact (leading to a 7-tuple). In the second case, the first three elements of the input are unified using domain coercion. In that case, the number of test cases is reduced drastically. All the numbers are shown in Table 4.1.

Rules	**3 Criteria**	**4 Criteria**	**Parallel**	**Parallel Unified**
1	$4 = 3^1 + 1$	$5 = 4^1 + 1$	$20 = 4 * 5$	14
2	$11 = 3^2 + 2$	$18 = 4^2 + 2$	$198 = 11 * 18$	77
3	$30 = 3^3 + 3$	$67 = 4^3 + 3$	$2010 = 30 * 67$	336
4	$85 = 3^4 + 4$	$260 = 4^4 + 4$	$22100 = 85 * 260$	1319
5	$248 = 3^5 + 5$	$1029 = 4^5 + 5$	$255192^1 = 248 * 1029$	4898

[1] projected value

Table 4.1: Number of generated test cases for several generic policies

Observing the coverage criterion we apply and its consequence on UPF policies, we can make two observations:

- We apply a very thorough coverage criterion, leading to an exhaustive testing of the specification.
- Considering the fact that real-word policies often consist of hundreds of rules, there will inherently be huge scalability issues.

While the achieved coverage is fine and well-suited for small-scale policies, the number of generated test cases (and the time complexity required for generating them) is unacceptably high for any real-world policy which may consist of hundreds of rules. In section 4.2, we present a solution that allows one to process these kind of policies as well, by providing a methodology to transform a policy specification.

Another, rather more ad-hoc possibility for improving the scalability, is to "hide" certain aspects of the specification from the test case generation algorithm. In the examples in this section, all definitions were unfolded. It is however possible to keep certain concepts closed, so the algorithm will not be able to perform a splitting upon them. This approach requires domain-knowledge, as the coverage is hereby explicitly reduced on certain aspects of the specification. We will further explore this effect in the firewall case study.

In certain cases however, it is desirable to enlarge the coverage, for example when one wants to consider so called boundary cases. A technique for doing this is also presented in section 4.2.

4.2 Policy Transformation Procedures

As has been presented in the previous section, the naive approach to test case generation does not scale well for large-scale policies. However, scalability is an important property for any practically-relevant test generation technique. The problem is not only due to our choice of the concrete test case generation algorithm, but is inherent: constructing systematic tests of security policies leads inevitably to very large cascades of case distinctions over input and output. The situation is worse if there is an underlying state evolving over time.

In this section, we present a technique to overcome this problem in many cases by using so called *verified policy transformation procedures*. In essence, a policy transformation procedure transforms a given policy p into a a semantically equivalent policy p', which is easier to test allowing for an improvement of the effectiveness of test case generation by up to several orders of magnitude. While not changing the coverage criterion explicitly, it will be applied to a transformed specification. Essential to guarding the security and guarantees when applying such an approach is the correctness of such a transformation procedure. Our theorem-prover based approach to modelling and test case generation allows us to prove formally the correctness of such transformations.

The general underlying idea behind this technique is the observation that a policy can be formulated in infinitely many syntactically different but semantically identical ways. We can take advantage of this fact by transforming a policy into a semantically equivalent formalisation, where the state-space can be limited. The technique is not new in the testing area, and is generally called *testability transformations* (see [HHH+04]). However, to our knowledge it has so far not been applied to model-based testing approaches, including security policy testing.

One might wonder why a semantically equivalent re-formulation of a security policy can lead to a smaller test case set which is moreover simpler to compute. The answer is fundamental to model-based (or program-based) test case generation: the precise form of a specification (or a program) does affect the test cases that are generated. For example, consider:

$$\text{if } x < 0 \ \vee \ x > 2 \text{ then } P \text{ else } Q$$

and

$$\text{if } x \in \{0, \ldots, 2\} \text{ then } Q \text{ else } P$$

The two statements are semantically equivalent. In the first case, many model-based test case generation algorithms (depending on the coverage criterion) will usually generate

the three test cases $x < 0$, $x > 2$ and $0 \leq x \wedge x \leq 2$. In the second case, however, only the two test cases $x \notin \{0, \ldots, 2\}$ and $x \in \{0, \ldots, 2\}$ would be generated. Here we assume that the procedure could not establish their equivalence.

As another example from the firewall policy domain, the semantics of the following two policies is equivalent. Here the rules have the obvious meaning:

$$\text{DenyFromTo } x\ y \oplus \text{AllowPort } x\ y\ p = \text{DenyFromTo } x\ y$$

However, for the policy on the left hand side test cases considering the port number p will be generated, but not so for the one on the right hand side.

Please note that for very simple scenarios, we will get policy transformations for free: the theorem prover could immediately determine the equivalence of the two policies above when creating the if-then-else cascade and removing the non-reachable parts. In almost all non-trivial examples however, manual intervention is required.

4.2.1 Technique of Policy Transformations

More formally, a policy transformation is defined as follows:

Definition 4.2.1 *A* **policy transformation** *for a policy P is a function taking P as input and returning a list of policies $[P_1, ...P_n]$, such that $P = P_1 \oplus ... \oplus P_n$, and where the domains of all policies P_i are pairwise disjoint, with the possible exception of P_n.*

Please note that in many cases n would just be one. This would for example apply to the scenarios above. The exception regarding the domain of the last element of the returned policies is justified by the fact that in most policies there is a default rule with the universal domain. Our techniques provide support for that case.

There are in general three kinds of strategies a transformation procedure might employ in order to ensure the transformed policy is better suited for test case generation:

1. Eliminate redundant rules: Rules which are never considered are not required and can be removed from the model.

2. Combine similar rules to a single one: For example, instead of a sequence of rules allowing one particular element, a single rule might allow a set containing all these elements.

3. Pre-partition the test space: Often, policies consist of several independent parts which might be processed individually.

The first point can be justified in almost all cases. For the second one, we must be aware that we reduce the coverage regarding the policy in some specific aspects when combining rules. The last point however, is very domain-specific and must be applied with care.

The reasoning above already leads to the conclusion that any concrete policy transformation strategy is inherently domain-specific. The Unified Policy Framework can merely provide the foundational and supporting concepts.

Smaller transformation procedures can be represented by a theorem. For example, the theorem

theorem $\mathrm{D}_U \oplus \mathrm{r} = \mathrm{D}_U$

can be added to the simplifier which will directly transform respective occurrences in a policy. This technique is however limited and not very generalisable, so further techniques are required.

In the UPF, a rule is a partial function. Many large-scale transformation procedures however, are better represented as *syntactic rules* operating on syntactic policy combinators. This choice paves the way for expressing the transformations process inside HOL by inductive and recursive functions manipulating abstract policy syntax. Alternatively, these transformations could have been implemented directly inside the prover by a so called tactic program. Defining the functions entirely inside HOL gives us the possibility to prove their termination and completeness formally.

Following this, a closed datatype defines the admissible rule operators for policy specification, and a semantic interpretation function provides the mapping to the original functional rules from the UPF. In the remainder of this section, we present the methodologies available to transformation procedures represented as syntactic rules.

The standard *map* and *fold* functions can be used to switch from the syntactic to the functional view of a policy. As abbreviation for transforming a list of policies into a single policy combined using the standard override operator, we define:

definition list2policy $\mathrm{l} = \mathrm{foldr}\ (\lambda \mathrm{x}\ \mathrm{y}.\ \mathrm{x} \oplus \mathrm{y})\ \mathrm{l}\ \emptyset$

Next we formalise the requirement of semantic equivalence for a transformation T on a policy p with the semantic interpretation function sem as:

$$\mathrm{sem}\ P = \mathrm{list2policy}\ (\mathrm{map}\ \mathrm{sem}\ (T\ P)) \tag{4.3}$$

This theorem states that the semantic interpretation of the policy P is equivalent to the policy created by application of the *list2policy* operator to a list of semantically interpreted policies created by application of the transformation T on policy P. To ensure correctness of the test case generation of a transformed policy, this theorem needs be proved correct for any concrete transformation procedure. It is either possible

to perform this proof for each given policy P anew, or to perform this proof once and for all for any given policy P (possibly under certain assumptions which will then have to be shown to hold when testing a concrete policy).

Next, we present how a transformation procedure can be integrated with testing. In order to achieve a pre-partitioning of the test space, the test specification must be rewritten into a set of test specifications, where each element considers only one part of the original specification. Technically, this can be achieved by rewriting the original test specification into a conjunction, which is later transformed into a sequence of subgoals, each processed individually by the test case generation algorithm. This technique is only correct if it can be assured that each test specification is creating test cases for the relevant domain only. From this fact stems the requirement that the domains of the subpolicies need to be pairwise distinct.

Recall that in general the format for a test specification is as follows:

P x $\Longrightarrow$SUT x = Policy x

First, we need the following function which transforms the conclusion of such a test specification into a conjunction of individual specifications if provided a list of policies.

fun SUTList :: ($\alpha \mapsto \beta$) $\Rightarrow \alpha \Rightarrow$ ($\alpha \mapsto \beta$) list $\Rightarrow$ bool **where**
SUTList SUT x (p#ps) = (x $\in$dom p $\longrightarrow$(SUT x = p x)) $\wedge$
(SUTList SUT x ps)
|SUTList SUT x [] = True

The following theorem is the most crucial one of the whole concept: it can be applied in a test specification to execute the transformation of the test specification into a conjunction of test specifications. Due to the fact that this is stated and proved as a theorem, the transformation is guaranteed to be correct, given the three assumptions in the theorem hold:

- The domain of the original policy must be universal (i.e., the policy must be complete)
- The semantical equivalence of the original policy with the list of subpolicies as stated in Equation 4.2.1 must hold
- The domains of the policies must be pairwise disjoint

theorem
assumes x $\in$dom P
assumes list2policy PL = P
assumes disjDom PL
shows (SUT x = P x) = SUTList SUT x PL

A variant of above theorem relaxes the last assumption such that the domain of the last element in the list of policies must not be disjoint from the others. For this case, an alternative version of *SUTList* exists, ensuring that the domain of the last subpolicy is the universe minus the domains of the other subpolicies.

Please note that the theorem establishing semantic equivalence of the transformation must be proven for this theorem to work. Also note that, in general, $PL = map\ sem\ (T\ p)$, however the theorem is more generic (there are transformations which do not need to operate on the syntactic layer and thus do not require a semantic interpretation function). The theorem itself does not mention the transformation procedure explicitly, but only its input and its result.

4.2.2 Distributivity of Transformations

UPF policies are very often created by a parallel composition of smaller policies. We have observed that this leads to a large state explosion hindering efficient test case generation. So the question arises how transformations can be used together with parallel compositions.

Of course, it is possible to devise a transformation procedure for a parallely combined policy. However, this is in many cases not easy achievable and not compatible with the goal of a modular modelling approach. Much better would be to have transformation procedures for the smaller policy parts and the means to transfer these transformations over parallel compositions.

Transformation procedures can be transferred over parallel compositions thanks to the fact that the law of distributivity holds over these operators, as already in section 3.6. In the following, we limit ourselves to the denial parallel operator, however the results apply to all four of them.

$(\mathrm{P1} \oplus \mathrm{P2}) \otimes_{\vee D} \mathrm{Q} = (\mathrm{P1} \otimes_{\vee D} \mathrm{Q}) \oplus (\mathrm{P2} \otimes_{\vee D} \mathrm{Q})$

In more detail, let us assume a policy under test being created as a combination of policies P and Q, and a transformation procedure transforming P into the list $[P_1, P_2, \ldots, P_N]$. Assume the policies are combined in parallel using one of the parallel operators:

$\mathrm{P} \otimes_{\vee D} \mathrm{Q}$

then, assuming semantic equivalence of the transformation, the following theorem holds:

$$(\mathrm{sem}\, P) \otimes_{\vee D} Q = \mathrm{list2policy}[\mathrm{sem}\, P_1 \otimes_{\vee D} Q, \ldots, \mathrm{sem}\, P_N \otimes_{\vee D} Q]$$

This gives rise to a transformation strategy for policies consisting of a number of parallely executed parts: define transformation procedures for the individual policies, which in

the end will lead to a transformed combined policy. Of course, the number of final subpolicies might grow very large when transformation procedures are being executed on several constituent policies, as all the subpolicies are paired. For this reason, it is sensible to have transformations which are similar in the sense that they create similar domains for their subpolicies. In that way, most domains of the combined subpolicies are empty, and those elements can be dropped.

4.2.3 User-defined Modifications of the Specification

So far, we have presented policy transformation procedures as a means to enhance efficiency of test case generation by reducing coverage of the original policy. Next, we show that they can also be used for the opposite: in some cases, one might want to increase coverage in certain areas. The most notable examples are boundary test cases. In essence, the idea of *boundary value testing* is defined "to choose test input values at the boundaries of the input domains. The rationale for boundary testing is straightforward: lots of faults in the SUT are located at the frontier between two functional behaviours." [UL06]

The test case generation algorithm we apply does not generate boundary values by default. As outlined below, transformation procedures allow one to add support for boundary value testing without having to change the test case generation algorithm. The technique is however not restricted to boundary value testing, but might be applied whenever one wants to enable domain-specific modifications of the test specification.

The underlying idea is to modify the test specification such that the desired boundary cases are also covered. This should be performed automatically and by a verified procedure, such that the test specification remains semantically equivalent. Thus the goal is *to transform the model of a policy into a semantically equivalent one where additional test cases for boundary cases are produced.* This however, is exactly what can be achieved by transformation procedures. Thus, the idea is to define *transformation procedures, which transform a policy into a semantically equivalent one additionally covering the boundary cases.*

As an example, take the following pseudo policy, allowing subject s to perform access a, but denying everyone else to do so. Here the subjects are modelled as integers, and we would like to add boundary tests for the subjects.

$$\text{AllowAccessForSubject } s\ a \oplus \text{DenyAccess } a$$

This policy is semantically equivalent to:

$$\text{AllowAccessForSubject } s\ a \oplus \text{DenyAccessForSubject } (s+1)\ a \oplus$$
$$\text{DenyAccessForSubject } (s-1)\ a \oplus \text{DenyAccess } a$$

However in the latter case explicit test cases for the boundary cases will be produced.

In most cases, there is an easy way how such rules can be added to a policy without changing its semantics: if there is a global default rule (e.g. D_U), any rule can be added after it using the override operator. As the domain of the default rule is the universe, they will never be applied. Note that in general only the domain of the additional rule is relevant, and we might either use a denial or an allowance rule. Most importantly, we do not necessarily have to know what the output of the original policy applied to the elements of the additional rule actually is.

4.2.4 Effect on Coverage

As we do not alter the test case generation algorithm, the coverage is not changed *directly*. However, we are changing the specification to be processed, and the coverage *regarding the original specification* is changed. As an example, we do no longer generate test cases for the rules which have been removed by the transformation. On the other hand, additional test cases will be generated for newly added rules. Additionally we reduce coverage regarding the original specification when combining several rules into a single one.

The pre-partitioning of the test space has the most significant implications on the coverage regarding the original specification. In more detail, what happens internally, is that the individual policies of the list will be processed individually. In other words, the test specification is decomposed into a list of test specifications, each with a smaller input space than the original policy. Recalling section 4.1, we know that in unit testing we have multiple condition coverage of the specification. After pre-partitioning the input space, this is reduced to a multiple condition coverage of each test specification by itself. The individual test specifications generated are completely independent from each other regarding test case generation. That is, the test adequacy criterion of the full policy is transformed into a set of test adequacy criterion for each subpolicy. While this separation of independent domains is often desirable, there are cases where it is not. So care must be be taken when partitioning the input space such that only really independent parts of a policy are being separated. In general, such a reasoning is not only policy-specific, but even application-specific and cannot be performed without considering the system to be tested against.

4.3 Sequence Testing

The second approach to testing policies of the UPF is the sequence testing approach, which we present in this section.

4.3.1 Definition

The sequence testing technique is used whenever one wants to test a state-based system. While not required for static policies, it is absolutely necessary for meaningful testing of stateful policy systems.

In our terminology, *sequence testing* can be defined as follows:

Definition 4.3.1 ***Sequence Testing*** *is the testing of a (policy-implementing) system where each test case consists of a sequence of access requests (or operations) and a sequence of observable outputs.*

A wide range of different approaches have been proposed for model-based testing of state-based systems, for example based on model-based languages such as B, Z, VDM, or on finite state machines, labelled transition systems, process algebras, etc. Here we present the approach we are using in testing models of the UPF. First, we outline how transition policies are represented as computations, how meaningful test specifications for sequence tests can be specified, and then provide indications of the workings of the test case generation algorithm that is run on such a model. Interestingly, both the test case and the test data generation algorithm do not have to be altered in order to accommodate sequence testing.

4.3.2 Technique of Sequence Testing

The object of consideration in sequence testing within the Unified Policy Framework (UPF) is a *transition policy* of the form

$(\iota \times \sigma) \mapsto (o \times \sigma)$

where ι is some kind of an input, o some (usually observable) output, and σ the (usually internal) state (see section 3.4). Such transition policies represent computations, and to reason over them, and more specifically to make them amenable to a test case generation algorithm within HOL, requires mechanisms to deal with states and state-transitions within the logic. However, HOL as a purely functional specification formalism has no such built-in mechanisms. Originally a concept from category theory, *monads* are one technique to overcome this limitation made popular for use in programming languages

by Moggi [Mog91] which is for example extensively used in the functional programming language Haskell.

Abstractly, a monad is a type constructor with a unit and a bind operator, enjoying unit and associativity properties. Due to well-known limitations of the Hindley-Millner type-system, it is not possible to represent monads as such in HOL directly. It is however possible to define concrete instances of monads, as it is done for example in [SB07]. In the following, we define a variant for state-exception monads, which model transition functions with preconditions, which is a very similar concept to transition policies. State-exception monads have the following type:

type_synonym (o, σ) M_{SE} = $\sigma \rightharpoonup$(o $\times\sigma$)

While there are other monad concepts available (for example state-backtrack monads or state-backtrack-exception monads), the state-exception monads are the closest to our concept of a transition policy. Using this concept, we can view a transition policy of type ($\iota\times\sigma$)$\mapsto$(o$\times\sigma$) as an i/o stepping function of type $\iota \Rightarrow$ (o decision ,σ) M_{SE}, where each stepping function may either fail for a given state σ and input ι, representing an undefinedness result of the transition policy, or produce an output o (including the decision) and a successor state.

These two representations are isomorphic and there are two conversion functions linking them. For example, to transform a transition policy into a state-exception monad, we can use:

definition p2M::($\iota\times\sigma$)$\mapsto$(o$\times\sigma$) $\Rightarrow\iota\Rightarrow$(o decision,$\sigma$)$M_{SE}$
where p2M p = $\lambda\iota\sigma$. case p (ι,σ) **of**
  $\lfloor$allow (o, σ')$\rfloor$ $\Rightarrow$ $\lfloor$(allow o, σ')$\rfloor$
| $\lfloor$deny (o, σ')$\rfloor$ $\Rightarrow$ $\lfloor$(deny o, σ')$\rfloor$
| $\bot\Rightarrow$ $\bot$

It is easy to check that p2M is a bijection to state exception monads.

The usual concepts of bind (representing sequential composition with value passing) and unit (representing the embedding of a value into a computation) over these monads are defined as follows:

definition $bind_{SE}$ :: (o,σ)M_{SE} $\Rightarrow$(o $\Rightarrow$(o',σ)M_{SE}) $\Rightarrow$(o',σ)M_{SE}
where $bind_{SE}$ f g = $\lambda\sigma$. case f σ **of** $\bot\Rightarrow\bot$
  | $\lfloor$out, σ'$\rfloor$ $\Rightarrow$ g out σ'

definition $unit_{SE}$ :: o $\Rightarrow$(o, σ)M_{SE}
where $unit_{SE}$ e = $\lambda\sigma$. $\lfloor$(e,σ)$\rfloor$

In the sequel, we will write x $\leftarrow$f; g for $bind_{SE}$ f(λ x. g) and return for $unit_{SE}$.

Using the introduced syntax, a computation sequence can be written as:

$o_1 \leftarrow f_1$; $o_2 \leftarrow f_2$; return (post o_1 o_2)

The standard monad identity and associativity laws are easily proved over these definitions:

- Left identity: x ←return a; k = k
- Right identity: x ←m; return x = m
- Associativity: y ←(x ←m; k); h = x ←m; (y ←k; h)

Computations applied on an input sequence can be modelled using a sequence-bind-operator, called mbind, defined as follows by recursive equations. This definition is fail-safe: in case of an exception, the current state is maintained. Additionally, note that the subsequent notions of test sequences allow the i/o stepping function (and the special case of a system under test) to stop execution *within* a sequence; such premature terminations are characterised by an output list which is shorter than the input list.

```
fun  mbind :: ι list  ⇒  (ι ⇒ (o,σ) M_SE) ⇒(o list,σ) M_SE where
 mbind [] iostep σ = ⌊([], σ)⌋
|mbind (a#H) iostep σ=
     case iostep a σ of
          ⊥   ⇒ ⌊([], σ)⌋
        | ⌊(out, σ')⌋ ⇒ (case mbind H iostep σ' of
                              ⊥    ⇒ ⌊([out],σ')⌋
                            | ⌊(outs,σ'')⌋ ⇒ ⌊(out#outs,σ'')⌋)
```

Using this operator, computation sequences of a system P on a sequence of inputs *is* can be written as:

os ←mbind is P; return (post os)

As a side remark, please note that this technique requires all the inputs and outputs to be of the same respective types. While this is usually the case for transition policies, in the general case we need to apply a technique called interface encapsulation that wraps up all input and output data into an own type.

A *validity* operator denoting successful computations satisfying a boolean predicate is also available:

```
definition _ ⊨_ :: σ ⇒ (bool,σ) M_SE ⇒bool
where σ⊨m = (m σ≠⊥∧ fst  (the (m σ)))
```

where *the* $\lfloor x \rfloor = x$.

The characterisation of the sequence-binding and validity operators is outlined in the following lemma:

lemma $\sigma \models$(s $\leftarrow$mbind (a#S) ioprog ; return (P s)) =
case ioprog a σ **of**
$\bot \Rightarrow$ ($\sigma \models$(return (P [])))
| $\lfloor$(b,σ')$\rfloor$ $\Rightarrow$ (σ' $\models$(s $\leftarrow$mbind S ioprog ; return (P (b#s))))

Using these concepts we can model test sequences together with the observable output. Next, we have to adjust the test specification to enable it for test sequence generation. The general recipe is to represent automata or labeled transition systems as (mutual) recursive acceptance predicates *accept* on input lists, i.e. *traces*, and the test case generation will attempt to explore the input lists used as stimulation of the system under test. Branching in an automaton will be represented by a disjunction, leading to a case-split in the test case generation. The test cases correspond to all paths through the automaton up to length d, determined by the depth parameter of the test case generation algorithm. The mbind-combinator takes care of the serialisation of the repetitive execution of SUT.

Essentially, the typical test specification can be paraphrased as follows: for all input traces t satisfying an acceptance predicate, if the modelled transition policy monad returns the output sequence x when t has been executed in a starting state σ_0, the system under test SUT should also return x.

Formally, this can be specified as follows, where $PolMon$ is the modelled transition policy monad:

test_spec
accept t $\Longrightarrow$
$\sigma_0 \models$(os $\leftarrow$mbind t PolMon; return (os=X)) $\longrightarrow$
SUT t σ_0= X

The predicate *accept* will henceforth be referred to as *test specification automaton*. This reflects the fact that it denotes a kind of automaton specifying all the acceptable input traces and because of its usage in a test specification. Technically, it is a predicate on traces, or, conceptually equivalent, a set of traces. However we explicitly refrain from providing a formal definition of the term, as there is a wide range of possibilities of how it might be represented, as long as it is some kind of predicate on input sequences.

Several variations of this general test specification concept are possible. For example, instead of only an acceptance predicate on the input trace, further predicates can be specified on both the input as well as on the output trace. The standard example is the requirement that the length of the output sequence should be the same as the one of the input sequence, denoting a successful computation. Additionally, the *return* part might contain further properties on both the output as well as on the input type than just $os = X$. Finally, the exact format of the last line ($SUT\ t\ \sigma_0 = X$) might also differ and is mainly dependent on the exact way how the tests are to be executed on the real

system (see section 4.4).

Another option, but less used in the UPF, also treats the *SUT* as a state-exception monad. This has, in general, the advantage that after generation of the test data (i.e. a real input trace), a test driver running the test sequence can be generated automatically, since the monad-operators as well as mbind and the validity operator are all executable.

The general methodology to process a test specification as above is to apply the test case generation algorithm on this specification using the definitions of the test specification automaton and its constituent parts, but to leave the transition policy monad intact. Consequently, this means that the test data set and the test coverage are only determined by the automaton, and the policy is irrelevant in this respect. This is in stark contrast to unit testing scenario introduced earlier. The policy itself is used after generation of the abstract test cases only, in order to calculate the outputs.

4.3.3 A General Example

In this section, we instantiate the general approach presented above with a small example policy with the goal of further explaining the methodologies underlying the sequence testing approach to policies specified in the Unified Policy Framework.

The setting is as follows: there are objects and subjects, both modelled as integers. Additionally there are four operations: create, read, write, and delete. All of them take a subject and an object as input.

The policy is as follows: only the creator of an object may read, write, and delete it. Creating an object is possible as long as it does not exist yet.

For modelling the policy, we define a state σ as a partial function from objects to subjects, storing the creators of the existing objects. Next, the predicate *acceptableOps* is defined for the allowed operations as outlined above:

fun acceptableOps :: (Operation $\times\sigma$) $\Rightarrow$ bool **where**
acceptableOps (read s obj, σ) = (σ obj = $\lfloor$s$\rfloor$)
|acceptableOps (Write s obj, σ) = (σ obj = $\lfloor$s$\rfloor$)
|acceptableOps (delete s obj, σ) = (σ obj = $\lfloor$s$\rfloor$)
|acceptableOps (create s obj, σ) = (obj $\notin$ (dom σ))

Using this predicate, the static policy can be defined using the standard UPF operators as:

rule1 = ({x. acceptableOps x} $\triangleleft$A$_U$) $\oplus$D$_U$

While the denial state transition ST_D is the identity, the allowance state transition ST_A changes the state after execution of a create and delete operation:

fun ST_A **where**
ST_A (create s obj, σ) = $\lfloor\sigma$(obj↦s)$\rfloor$
|ST_A (delete s obj, σ) = $\lfloor\sigma$(obj:=⊥)$\rfloor$
|ST_A (x, σ) = $\lfloor\sigma\rfloor$

Again, standard UPF operators can be used to combine the subpolicy and the state transitions to a transition policy, which itself is transformed into a state-exception monad:

definition policy = p2M ((ST_A,ST_D) $\otimes_\nabla$ rule1) o (λx. (x,x))

Note that the output of the monad is unit only.

As outlined above, we may model acceptable input traces as automata for creating test input sequences. Even when considering this artificially simple example, it becomes apparent why a test case generation for transition policies necessarily needs to be constraint by such an automaton in order to be meaningful. The number of possible test sequences of length n that represent all possible interleavings of the four operations is already very large. If there is additionally a splitting to be performed inside the operations (which is usually the case), this grows even larger. A random process picking a few traces is unlikely to lead to a satisfying testing coverage or to high-quality tests.

A test specification automaton does not necessarily need to be specified completely, and actually in most cases is not. In the sequel, we present a couple of typical automata specifications and explore the corresponding effects on test case generation.

A very constrained automaton defines sequences of length three, where the first is a create operation, the last a read, and the intermediate one either a write or a delete. All of them are executed by the same subject on the same object. Such an automaton is defined as follows, where the *Si*s are members of a datatype denoting the internal states of the automaton:

fun test_trace1 **where**
test_trace1 S1 s o (x#xs) = (x = create s o ∧test_trace1 S2 s o xs)
|test_trace1 S2 s o (x#xs) = (x = Write s o ∨x = delete s o
∧ test_trace1 S3 s o xs)
|test_trace1 S3 s o (x#xs) = (x = read s o ∧test_trace1 S4 s o xs)
|test_trace1 S s o x = (S=S4 ∧x = [])

And the following definition defines the set of all traces specified by this automaton.

definition
trace_set1 s o = {x. (test_trace S1 s o x)}

The corresponding test specification is as follows. Note that in all examples in this section we start the tests in an empty state.

```
test_spec
⟦length t = length X; ∃ s o. trace_set1 s o t⟧ ⟹
∅ ⊨(os ←mbind t policy; return (os=X)) ⟶
 SUT t = X
```

That is, we want to generate test sequences denoting successful runs of the test specification automaton, with an arbitrary subject and object. Additionally, we require the output sequence to be of the same length as the input sequence. As noted in the last section, the standard way to process such test specifications is to unfold the definitions denoting acceptable input traces, but not the transition policy monad for the test case generation algorithm, and afterwards calculate the desired outputs.

Not surprisingly, two test cases are generated: one for both possible intermediate operations. No special cases are generated for different objects and subjects, as which ones are chosen is not relevant regarding the test specification. The test data will look as follows. Note that the concrete values of the integers denoting the subject and object might differ, as there is a random solver being executed as part of the test data generation algorithm and the values are not constrained.

```
SUT [create 6 2, Write 6 2, read 6 2] = [allow (), allow (), allow ()]
SUT [create 9 4, delete 9 4, read 9 4] = [allow (), allow (), deny ()]
```

A more interesting test specification automaton is a slight extension of the previous one:

```
fun test_trace2 where
 test_trace2 S1 s o (x#xs) = (x = create s o ∧test_trace2 S2 s o xs)
|test_trace2 S2 s o (x#xs) = (((x = Write s o ∨x = read s o)
                                  ∧ test_trace2 S2 s o xs)
                               ∨ (x = delete s o ∧ test_trace2 S3 s o xs))
|test_trace2 S s o x = (S=S3 ∧x = [])
```

The difference now is that the length of the sequence is not restricted: there is an arbitrary number of the two possible intermediate operations. For the test case generation algorithm to terminate, a bound must be specified. A regularity hypothesis is then generated specifying that the test specification holds for all sequences exceeding that bound given it holds for all sequences below this bound (see section 2.2). In Table 4.2, we have generated test sequences according to that automaton with an increasing upper bound up to length 10, and show how many test cases are generated for each case. The number of test cases doubles approximately when increasing the maximum length by 1. This can explained by considering that when increasing the length, two alternative messages can be included in any of the previous sequences. In more detail, if we fix the object and the subject of a trace, the number of fixed elements of the sequence is two (the first and the last message), while there are two possibilities for the remaining elements. Thus, for any given length n, with $n > 2$, the number of possible traces is 2^{n-2}. Finally, the number of possible input sequences for a *maximum* length n is:

Depth	3	4	5	6	7	8	9	10
Test Cases	3	7	15	31	63	127	255	511

Table 4.2: Number of test cases for a given depth

$$f(n) = 2^{n-1} - 1 \tag{4.4}$$

Here, the number of generated test data is equal to the number of possible abstract input traces. Currently, no splitting is made on the variables of the input operation, i.e. on the concrete subject and the object of the trace. This extension is made next.

So far, we have restricted the potential concrete objects and subjects of the input trace to one single, but arbitrary instance each. Next, we extend this behaviour to allow several instances to be generated. The first possibility would be to leave them completely open, that is they do not need to coincide within the test trace. If done this using the test specification above (*test_trace2*), we would still end up with the same number of generated test cases. However, this time it would be highly likely (depending on outcome of the random number generator) that the operations will operate each on different objects and subjects, will consequently be uncorrelated, and only the first operation would be allowed. In most scenarios, such tests would not be desired and lead to a low coverage.

A common approach is to allow a fixed number of subjects and objects, which can either be set to a specific value or be arbitrary. In the following example, the subjects of the operations must be members of a specific set, while the object is still unique among all operations in the trace.

```
fun test_trace3 where
 test_trace3 S1  s o (x#xs) = ∃s1∈s. x = create s1 o ∧test_trace3 S2 s o xs
|test_trace3 S2 s o (x#xs) = ∃s1∈s. ((x = Write s1 o) ∨(x = delete s1 o))
                                    ∧ test_trace3 S3 s o xs
|test_trace3 S3 s o (x#xs) = ∃s1∈s. (x = read s1 o) ∧test_trace3 S4 s o xs
|test_trace3 S s o x = (S=S4 ∧x = [])
```

Using the same definition for modelling the acceptable set of traces, the test specification for such an automaton can be rewritten to:

```
test_spec
 ⟦length t = length X; s = {s1,s2}; ∃ o. trace_set3 s o t⟧ ⟹
 ∅ ⊨(os ←mbind t policy; return (os=X)) ⟶
SUT t = X
```

In this example, two subjects are allowed. Note that they are still arbitrary, and do not

Subjects	1	2	3	4	5	6	7	8	9	10
Test Cases	2	16	54	128	250	432	686	1024	1458	2000

Table 4.3: Number of test cases for a fixed number of subjects

even necessarily have to be distinct. In Table 4.3, we present the number of generated test cases for an increasing number of possible subjects to be allowed. Obviously the number is increasing quite quickly. This is due to the fact that the algorithm will try to generate all possible interleavings of the operations. In more detail, there are three input elements in the trace. The number of possibilities for the first and the last one is n, where n is the number of admissible subjects. As the second element allows two operations, the number of possibilities in that case is $2 * n$. This indicates that the number of possible input traces is $2 * n^3$, coinciding with the numbers given in Table 4.3.

4.3.4 Coverage

In the last section, we already provided some indications about the achieved coverage with the help of a small but general example. Here we would like to report further on it and draw more theoretic conclusions. First, we need to recall the quite different approaches to testing in the unit and in the sequence case:

- In the *unit case*, we explore all the paths of the policy to generate the test cases.
- In the *sequence case*, we define a test specification automaton, specifying the set of acceptable input traces, according to which the test sequences are generated. In other words, the concrete structure of the policy is not relevant for creating the inputs of the test data.

Remember that a common strategy will be to combine both testing approaches; they are not completely separate from each other.

For the reasons above, we do not speak about coverage regarding the policy when performing sequence testing, but rather about coverage regarding the test specification automaton. This immediately leads to the observation that the achieved coverage is even more user-adjustable than in the unit case, as the choice of the automaton determines the desired coverage. In the literature, this approach has sometimes been called *explicit test case specifications* (for example in [UL06]), as it explicitly allows one to specify the kinds of tests one wants to generate.

If we consider coverage regarding the test specification automaton, we might apply any transition-based coverage criteria. In essence, in sequence testing we perform a combi-

natorial testing on the input sequences. That is, *all possible abstract traces which are allowed by the test specification automaton are generated*. The closest coverage criteria conforming to this requirement called is ***all-paths coverage, modulo regularity***. All-paths coverage requires that all paths of the transition system must be traversed at least once [UL06]. In the presence of loops, this might lead to infinite sequences. Therefore we additionally apply the regularity criterion which specifies the maximum length of a sequence, and consequently giving an upper bound of iterations of loops.

4.3.5 Effect of the Coverage on Scalability

In this section, we report of the effects of the coverage criterion used on general UPF policies. We will explore the kinds and number of test cases generated regarding standard automata constructs of the UPF.

In almost all interesting test specifications, the automaton will not be completely specified. The three main factors often being left under-specified are:

1. Some input elements are only partially specified. This corresponds to the scenarios from the previous section where on the one hand, in one state two operations can be executed, and on the other hand one in some scenarios several subjects were allowed for each operation.

2. From some states, several transitions are possible. This corresponds to the first extended scenario from the previous section, where the second state might be passed an arbitrary number of times.

3. Some states of the automaton are completely unknown. This corresponds to a case where for example only the start and the end of the input sequence are (at least partially) specified, but nothing is known about the intermediate states.

Note that some instances of under-specifications actually correspond to a non-deterministic automaton. While a completely specified and deterministic automaton only allows one trace, and consequently will lead to only one test case being generated, all sources of under-specifications will lead to an increased number and thus also a larger coverage of the tests on the system.

Regarding the first case, where some parts of the input elements are partially left unspecified, the calculation of the number of generated test cases is the easiest. Let $x_1, ...x_n$ be an admissible input trace, and k_i be the number of possible abstract instantiations for x_i admitted by the test specification automaton A. Then the number of test cases (T) generated according to A can be calculated by the following formula:

$$T1 = \prod_{i=1}^{n} k_i \tag{4.5}$$

Which simplifies to k^n if all k_is are equal. Note that in full generality, the calculation of k is not trivial without detailed knowledge of the test case generation algorithm, as it has been described in section 2.2.

In the second case, where several transitions are possible, we need to distinguish two cases. The first alternative applies to the scenario where a transition from state s might either lead to the state s again or to some state s' different from s. This case corresponds to the first extended scenario in the previous section. Its consequence is that the length of the input sequence is now not fixed. For each given length l, we might define a sub-automaton from A_l only allowing traces of length l. Let $T1_i$ be the number of generated test cases according to automaton A_i for all test sequences of length i, as calculated by Equation 4.5, and m the depth of the test case generation algorithm. Then the number of test cases according to A is:

$$T2 = \sum_{i=1}^{m} T1_i \tag{4.6}$$

The second alternative of the second case applies whenever from one state s, two or more transitions to states distinct from s are possible. Such an automaton can always be transformed to one where this case does not happen (c.f. standard procedures to transform a non-deterministic automaton into a deterministic one). After this transformation, Equation 4.5, possibly combined with the calculations below, applies again. An alternative - and often simpler - way, is to construct several sub-automata from the big automaton, where in each sub-automata, there is only one transition possible in each state. Let $A_1, ..., A_n$ be the sub-automata which were constructed from A, and $T2_i$ be the number of test cases generated by sub-automaton A_i, according to Equation 4.6. Then the number of generated test cases according to A is the sum of the number of generated test cases of the sub-automata:

$$T3 = \sum_{i=1}^{n} T2_i \tag{4.7}$$

The last case, where some parts of the automaton are completely unknown, is more difficult to assess in general. Basically the two formulae above can still be applied, however it is more challenging to determine k, which this time heavily depends on the domain-specific parts of the underlying testing theory. In some cases, for example, the intermediate elements might be left completely undetermined, in which case k would only be one, while in other cases the inputs could be elements of a datatype in which case k would be equal to the number of elements in this datatype.

As a summary of this section regarding coverage, the findings of section 4.1 can be extended. While when generating unit tests for a static policy, (almost) all possible decision paths are generated, here (almost) all possible traces as defined by the test specification automaton are generated. That is, in general, we attain a good coverage of the test specification. In order for the tests to be generated efficiently and having a good

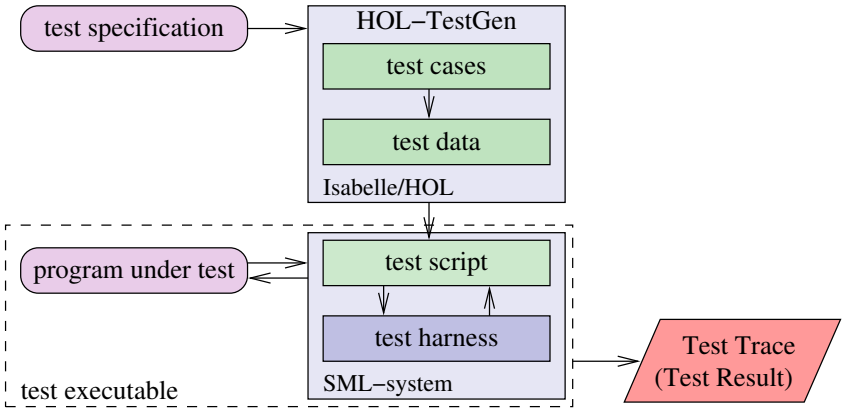

Figure 4.1: Overview of the Standard Workflow of HOL-TestGen

quality of the test data set this requires one to constrain the specification. Choosing good test specification automata and deciding on the depth of the test space exploration cannot be solved in general, as it is highly domain-specific, and depends on the concrete purpose of the tests to be executed.

4.4 Test Execution

So far we have been concerned with modelling systems and policies and how to generate test data according to these models. In this section, we outline how this test data can be used for execution of the tests. One advantage of model-based testing over other testing approaches lies in the possibility to not only automate generation of test data, but also the automation of the execution of the tests. The reason is that we already have a model of the system, and if that is provided in sufficient details, test execution can be automated in large parts. To fully automate this, either the code of the system under test must be provided, or there must be some public interface which can be called automatically. Remember also that in model-based testing the models are not only used for generating the inputs to the tests, but also to serve as oracle providing the expected outcome of the tests.

In general, there are four possible ways to execute tests in our framework:

- For white-box unit tests of local applications where the code of the system under test is available, HOL-TestGen can export a test script. This is shown in the lower part of Figure 4.1. The generated test script is an SML script that, together with a test harness provided by HOL-TestGen, can be executed independently from HOL-TestGen using an arbitrary SML compiler. By exploiting the various foreign language interfaces of the different SML compilers, this allows for an automated setup for testing implementations in programming languages such as Java, C, SML,

or any language running on the .NET environment. As we are mostly dealing with black-box testing scenarios in our framework, this approach is rarely applicable.

- Human-readable test script generation. In this scenario, the test data is exported into a file which can be used by a test engineer to manually execute the tests. Alternatively, the file could be in a form usable for importing into any standard testing software, facilitating test execution.

- Domain-specific test execution tool. In this scenario, a possibly domain-specific tool takes the generated test data and possibly some system description as input and executes the tests on the system. Such as a solution is highly domain-specific. We provide an instance of this for the firewall domain.

- Automated test script generation executing the tests on a standardised interface. This scenario is available whenever the system under test provides a standardised way of being accessed. As a notable example, we have developed the techniques for testing systems conforming to WSDL-based web services.

In all three scenarios, there remains the problem which is inherent to model-based testing: the mapping of the model concepts to the concepts of the system under test. A model is usually on a more abstract level than the implementation, and the types do often not agree. An application-specific mapping step thus has to be performed in any case.

4.5 Summary

In this chapter, we have reported on methods to perform model-based testing of conformance of systems to a policy specified in the Unified Policy Framework (UPF). The test generation can be grouped in two distinct categories: unit tests where we test single requests to the system, and sequence tests where sequences of requests are being executed. Both categories are performed in a rather distinct way. While in the first case, we achieve a thorough structural coverage of the policy which can be seen as a big if-then-else cascade, we work with a concept called test specification automata in the latter. These automata are description of executional behaviour and are used to generate the desired test sequences. As policies are in general very large, any policy testing technique inherently has scalability issues. We provide a partial solution by an application of the concept of testability transformations to policies, called policy transformation procedures. The presentation is concluded with an overview of the different possibilities for test execution.

5 Modelling and Testing the Information Governance Policies of the NPfIT

In this chapter, we introduce a significant large-scale instantiation of the Unified Policy Framework with the domain of access control to health care records. In more detail, the domain consists of the information governance policies of applications being part of the National Programme for Information Technology (NPfIT) in the NHS in England.

From the modelling side of view, the NPfIT domain is very well suited for a case study. The security policies as presented below are very diverse, complex, interesting, but also challenging to model. Furthermore, they have been created mostly by people from the health-care domain rather than by security researchers only. As such, the concepts nicely present what is actually expected to be required in the real world. We believe that any framework where the policy concepts of the NPfIT can be modelled is powerful enough to cater for a very wide range of policies as encountered in practice. In general, the concepts are a typical example of a complex and realistic access control policies, consisting of a range of different policy concepts. Thus, the instantiation allows us to present a typical usage of the UPF and of our model-based policy testing framework and proves its applicability in real-world scenarios.

5.1 The National Programme for IT in the NHS

5.1.1 Introduction to the Problem Domain

The National Programme for IT in the NHS in England is a very large-scale project aiming to modernise the complete IT infrastructure in the English NHS. It consists of a couple of national services and a large number of local applications deployed in hospitals, GP practices, pharmacies, etc. An important part of the programme is the Care Records Service (CRS), providing Electronic Health Records of every patient in England. These records exist in two versions: the detailed one which is held locally, and a Summary Care Record (SCR), which is held nationally, in a system which is called the *Spine*. The Spine also provides a couple of other national services provided for the local applications, for

example an access control framework. Regarding the local level, there is often a range of applications for each task and local NHS entities have some choices in deciding which application they want to use. However, they all need to integrate with the Spine and the other national services and conform to their specifications. In total, there are hundreds of applications which are part of this project.

In the sequel, we will focus on one particular part of the programme: access to the Summary Care Records held on the Spine, provided by a local application. The rights of the information subject to privacy, and the need to provide an efficient and effective service to the customer (who is often also the information subject) pose conflicting information access requirements, and defining policies that balance the two is a challenge. This holds in particular for applications in the health care domain in which very sensitive data is handled, but withholding that data may endanger the patient's health. For example, an emergency paramedic could harm an accident victim by administering emergency treatment without knowing about allergies and other medication being used. The general principles underlying who may access which data, are called the *Information Governance Principles*

It must be noted that while the project was very ambitious in the beginning and has been called the largest civilian IT project in the world, it is mostly considered a failure as of today. Its implementation has been accompanied by delays, cost overruns, and significant changes to the specifications. For this reason, officially the project has been abandoned. Nonetheless, even though the project as such is no longer alive, a very large number of systems the project consisted of have been delivered and are in use accross the country, or will be delivered in the future. But most importantly is that no application does actually implement the policies as originally specified and as presented in this thesis. We present the concepts of the policies the way they have been specified at some point during the project, even if later they have been removed or simplified. The reason is that we strongly believe that the policy concepts are indeed valuable as such, will be used in closely related forms in other systems (not restricted to the health-care domain) and are not the reason for the failure of the project.

Abstractly speaking, the problem domain is the following:

- Ensuring conformance of applications to high-level policy guidelines
- A large number of legacy applications to be integrated
- The configurations are often performed by people not having access to the source code of the applications but have to provide evidence that the systems comply with regulations.
- System being delivered by a large number of suppliers
- A considerable amount of flexibility required while still maintaining overall integrity

- Very large complexity, both in terms of size and complexity of single rules
- Break-glass mechanisms and similar (ensuring access is granted in emergency situation)
- No point-of reference for policy: diverse configurations in place

Compliance of health-care applications with the NPfIT IG policy is a good example of where model-based testing could usefully augment the IT governance toolset:

- non-compliance has serious implications in terms of patient privacy and potentially safety, and may leave the service and software providers exposed to prosecution and litigation;
- the policy is complex and subject to on-going change;
- it applies to a range of application types from multiple providers.

5.1.2 Information Governance in the NPfIT

As can easily be imagined, the access control policies - governing who may do what in these systems and who may access which data - of such a large systems are very complex. While there do exist national guidelines of how access control should be done, it is the responsibility of each individual application to implement the policy correctly. The Spine merely provides the necessary information required to make the policy decision.

Note that in the following, the term *user* denotes an NHS employee.

To ensure that users can only execute functions and read the data that they need for the care of the patient, several elements are used:

- RBAC: users only get access to those functions necessary, according to their job profile.
- Legitimate Relationships: only users who are involved into a patient's care get access to health data.
- Patient's Consent: patients can choose how much of their data can be shared.
- Sealed Envelopes: users and patients can hide some of the data from some specific users or from all users.

There is no definite specification of a policy which all applications have to conform to at a certain point in time. Rather, the concepts as presented below are in some points just vague guidelines, and different applications will have to conform to different versions of the policies. The only part which is more standard is RBAC. The Legitimate

Relationships have so far never been implemented to their fullest extent, the Patient's Consent concept has been redefined significantly, and the concept of Sealed Envelopes has been dropped before actually completely being specified in the details.

We believe that model-based conformance testing based on the Unified Policy Framework can be a very valuable approach towards increasing the quality of the applications in this specific area, mainly due to the following reasons:

- A verification approach is impossible to be successful in such a complex scenario with a heterogeneous and distributed code-base.
- Integration of systems delivered by a large number of different third-parties.
- There is a set of different concepts in different versions. Having a modular approach as in the UPF allows quick modelling of the different scenarios.
- Having a model-based, automated approach may increase assurance of the authority and the public into the quality of the applications, even if the internals of the systems need to remain confidential.

Next, we present the four individual access control concepts in more detail.

RBAC

The NPfIT does not use the term RBAC using its standard semantics. Rather, they use some kind of Administrative Role-based Access Control (ARBAC). In general, RBAC in the context here is concerned with controlling which users can have access to which application functions, based on the job they do, and consequently only indirectly what kind of data they can access.

In traditional RBAC, the access control relation is divided into two separate relations: first, users are mapped to roles, and another mapping defines the rights of each single role. Usually, roles also have a hierarchy imposed upon them, where smaller roles inherit all the permissions of the roles above them in the hierarchy. The RBAC used in the NPfIT extends this model in several ways. First of all, users are not assigned a role, but one or more *User Role Profiles (URPs)*. These are granted by *User Sponsors*, and each user can be assigned several URPs. As only one of them can be active at one time, the user has to specify which one to use after each log-on. This concept is similar to standard RBAC sessions.

Each URP permits the user to perform several *Activities*. Activities are generic descriptions of business functionality. Each application that is part of the programme, must define its set of application functions which should be controlled by RBAC. For each of these functions, it defines a mapping to one or more activities. So each permitted activity gives a user access to a set of application functions, different for each application.

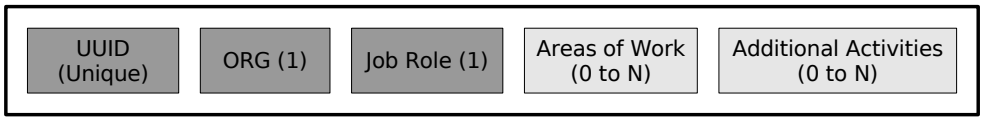

Figure 5.1: User Role Profile URP

A URP (see Figure 5.1) contains the following elements:

- **UUID:** Unique User ID.
- **Org:** the organisation on whose behalf the URP is being granted.
- **Job Role:** Exactly one in each URP. A generic description of the job of a user, e.g. Nurse.
- **Areas of Work:** From 0 to N. Some areas of work, in combination with a specific role, confer additional access rights. They were only rarely used and are supposed to eventually be removed from the programme.
- **Additional Activities:** From 0 to N. They are used to add some flexibility and to keep the required number of roles as small as possible. They grant a user additional permissions which he or she otherwise would not have.

The last three elements are also called *attributes*. They govern which activities are allowed for a given URP. There are three possibilities why an activity could be granted:

- Most roles allow some activities
- Some combinations of roles and areas of work allow further activities
- All activities in the set of additional activities are granted directly

Recapitulating, to get all permitted application functions given a URP, first, the URP is mapped to a set of activities, and, second, each activity is mapped to some function. Additionally, there is a hierarchy defined on the activities, which facilitates policy management. Child activities are included in their parent, of course transitively.

The mappings between the roles and the activities, including the areas of work, as well as the hierarchy on the activities and the definitions of the different attributes are stored in the *National RBAC Database (NRD)*. In other words, this database contains the set of activities granted for each role and role-area of work combination. These mappings are defined nationally and thus common throughout the programme. Local organisations, cannot change them. They can only influence the policy via the assignment of URPs and via the mapping to an application's functions.

Legitimate Relationship (LR)

Users who, according to the RBAC policy and their URP, are allowed to use application functions that access health care records should not be allowed to access the data of every patient throughout England, but only those whose care they are involved in. This is enforced by the concept of Legitimate Relationships (LRs). An LR allows a user to access a care record of a patient with whom he has a valid relationship.

To facilitate LR management, the users are grouped into workgroups, which reflect the organisational structure of a workplace. A user can access patient data only if he or she belongs to a workgroup having an LR with the patient in question. Workgroups are ordered in a hierarchy and are managed by the individual organisations of the NHS. LRs are generally between a patient and a workgroup, even though some are only between a patient and a user.

LRs are dynamic: they are created, removed, transferred, and carry an expiry date. They are thus part of a dynamic state. Additionally, an LR can also be frozen; this means that a user will only get access to the data created before a specific date.

There are ten different kinds of LRs: eight of them are between a patient and a workgroup and two between a patient and an individual user. There is a complex set of rules governing the dynamic behaviour of these LRs. Currently, the following LRs exist:

- Patient Referral
- Patient Self-Referral
- Patient Registration
- General Practice Registration
- Subject Access Request
- Patient Complaint or Ligitation
- Expressed Patient Consent to Access
- Court Order or Other Legal Demand
- Self-Claimed
- Colleague-Granted

The last two are between a patient and a single user only. As an example of a specific rule about the behaviour of the LRs, consider the following one:

> *The care professional granting [a colleague-granted] relationship will only be able to do so if they already have a Legitimate Relationship with the patient, and that relationship must not have been self-claimed. [NHS06]*

While it is clearly defined under what circumstances Legitimate Relationships can be created, this is done by the individual applications themselves, usually transparently for a user as part of the workflow within the applications. For example, when a GP refers a patient to a hospital clinic, the recipient application will automatically create an LR for

the workgroup associated with the clinical team. The applications report the changes in the Legitimate Relationships to the Spine, which stores the user accounts including the URPs, the workgroups and their hierarchy, and also all the current LRs. Applications can query that service to find out if a user has an LR with a patient.

Patient's Consent (PC)

Patients possess some freedom in their choice of how much of their data will be made available on the Spine. The original model was that there was one flag for each patient stored on the Spine. This flag had three states: unknown, consent, dissent, where unknown was treated as consent. In the case of a dissent, users in an organisation will not be able to access the data created by someone from another organisation. However, in case of a referral, the care record will usually be shared automatically.

In the adapted model, the default flag is unknown, with a semantics of "ask me every time if I want to upload that data". Dissent now means no writing, so the data will not even be uploaded to the Spine.

Some of the data will always be shared, even if users dissent. This includes all the data stored in the *Personal Demographic Service (PDS)*, for example demographics and information about a patient's GP. The consent model is only valid for data stored in the *Personal Spine Information Service (PSIS)*. The Summary Care Record is the combination of the two.

Sealed Envelope (SE)

Users are able to limit access to sensitive information within patient records. This process is called *sealing*. There are three types of sealing:

- seal: hidden from everybody except from users in a workgroup explicitly granted access in an accessor list. Other users see that something is sealed here and can override the seal in specific cases.
- seal and lock: this data can only be seen by the author. That data will never be shown to others, actually other users are not even aware of its existence.
- seal from the patient: users may see the data if they explicitly choose to do so, but must not show it to the patient. These entries are by default not displayed, in case the patient watches the screen of the user.

Patients cannot create, edit, or remove the seals themselves. They have to ask a user to do this on their behalf. Some data are sealed automatically. This includes for example some test results. The sealing process itself is not enforced via encryption, only via

access control mechanisms. It is assumed that all applications accessing the Spine are trusted.

While Sealed Envelopes (SEs) are a powerful policy concept with many usage possibilities outside of the health care domain as well, they have also added considerable additional complexity. For this reason, as well as for saving costs, they have been removed from the programme. We do still include them in the models, as we believe in the general value of the concept as such in other scenarios.

Additional remarks on the policy

The complete policy is the integration of all of the presented concepts. While the Spine stores and delivers all the required information, it is the responsibility of each single application to conform to these principles correctly. Therefore, each action of each user has to be controlled and audited.

It is in hardly possible to specify a policy for more than a million users in such a diverse setting that will always provide a decision which is considered "correct". First because of the size and complexity of such a policy, but also because not all future events can be foreseen. This is especially true for a health care scenario. For example, one would want a paramedic to have access to a patient's record in case of an accident. Also, legal issues can be a reason why an access control rule should be temporarily discarded. Policy mechanisms which allow one to circumvent default policy rules are called "break-glass" mechanisms. In the NPfIT, authorised users always have the right to access data. However, the application must clearly warn them when they try to access data they are not normally entitled to, the user must provide a valid reason for his behaviour (for example an emergency or a court order), the breaching of the rules needs to be reported, and approval must be given afterwards by the organisation's Caldicott Guardian (members of staff with a responsibility to ensure patient data is kept secure, each NHS organisation needs to appoint one). This concept is related to the concept of obligations.

5.2 Modelling the NPfIT Policies

In this section, we present how we modelled the information governance policies of the NPfIT as an instantiation of the UPF.

The challenges in *modelling* the NPfIT policies were manifold:

- The access control rules for patient-identifiable information are complex and reflect the trade-off between patient confidentiality, usability, functional, and legislative constraints. Traditional discretionary and mandatory access control models as well

as standard RBAC [RBA] are insufficiently expressive to capture complex policies such as LR, SE, or Patient's Consent (PC) management (presented below). Therefore, both a framework for their *uniform* modelling as well as their combination is needed (as provided by the UPF).

- The access rules of such a large system comprise not only elementary rules of data-access, but also access to security policies themselves enabling *policy management*. The latter is, for example, modelled in administrative RBAC [SBM99, RBA] models.

- The requirements are mandated by laws, official guidelines and ethical positions (e. g. [The97, Dep03]) that are prone to change. Such changes have to be enforced throughout a distributed and heterogeneous system. Moreover, the accessing applications also have to conform to local policies.

Abstractly speaking, the modelling strategy is the following:

- Model the generic concepts of the scenario, here health records, the desired system operations, etc.

- Model the different policy parts in small units, here typically using a policy type of the form $(\iota \times \sigma \mapsto \text{unit})$, with different kinds of states σ.

- Model the system behaviour, again in a modular way, leading to two automaton: one for the normal behaviour (i.e. the state transitions employed if the policy grants an access), and one for the exceptional behaviour (i.e. the state transitions employed if the policy denies access).

- Use the combinators of the UPF to combine these parts into a transition policy. According to the desired test scenario, different combinations might be desired.

- Transform the combined policy into a state-exception monad to enable use of the sequence testing techniques of our framework.

In the following, we first present how we modelled the general concepts, then the individual sub-policies and state transitions, and finally show their combination. As mentioned before, there does not exist one single specification of the policy as such. For this reason, we picked the general concepts of each individual policy aspect and formalised for each kind of rules one or several typical examples. This was especially necessary for the concept of Sealed Envelopes, which has been removed from the project before it was completely specified. RBAC on the other hand, which is in use at the NPfIT for several years as of yet, was formalised completely for one specific, widely deployed application (eSAP) using NRD version 25.

5.2.1 General Concepts

First, we describe how we model the main concepts of the NPfIT system that are relevant for the policy.

URPs and Attributes

Each user is given one or more User Role Profile (URP). Every time a user wants to execute an access-controlled function, he needs to present one of his URPs. A URP consists of a user, an organisation, a role, a set areas of work, and a set of activities. All of these concepts are modelled in the sequel.

Users are modelled as a type synonym to integers. This conforms to the reality, where each NHS employee is given a unique NHS id. The same holds for the organisation. The organisation stored in a URP is the organisation that issued this URP.

type_synonym user = int
type_synonym org_id = int

The *activities* are modelled as integers. Internally by the NHS, they are referenced by a number, and we use the same numbers. Additionally, we provide definitions for each activity such that they can also be referenced by their name, and provide a set (called *activities*) containing all valid activity numbers (not all integers denote a valid activity). There are currently 151 activities, following is a small selection of them:

type_synonym Activity = int

definition ViewSummaryHealthRecords = 0370
definition ViewPatientMedication = 0401
definition ViewDetailedHealthRecords = 0360
definition RecordPatientMedication = 0069
definition PerformClinicalCoding = 0790
definition PerformDetailedHealthRecord = 0380
definition VerifyHealthRecords = 8028

The number of *roles* is much smaller. Currently there are 21 of them. This allows us to define them as a datatype. The following roles are available. Please note that unlike in most traditional RBAC flavours, there is no hierarchy on the roles.

datatype Role =
RegistrationAuthorityAgent | RootRegistrationAuthorityManager
|RegistrationAuthorityManager | CaldicottGuardian
| ClinicalPractitioner | Nurse
|HealthProfessional | HealthcareStudent

|Midwife | Bank
|BiomedicalScientist | MedicalSecretary
|ClinicalCoder | AdminClinicalSupport
|SystemsSupport | PrivacyOfficer
|Receptionist | Clerical
|InformationOfficer | HealthRecordsManager
|SocialWorker

Finally, areas of work are also modelled as integers.

```
type_synonym AoW = int
```

Having all the ingredients ready, a URP can be defined as a combination of them, using the *record* concept. A record of n fields is essentially an n-tuple, but the record's components have names, which can make expressions easier to read and reduces the risk of confusing one field for another. Accessor and update functions are automatically defined when defining a record.

```
record urp = nhs_id     :: user
             org        :: org_id
             role       :: Role
             aows       :: AoW set
             activities :: Activity set
```

As an example of how concrete instances of URPs can be defined in the employed notation, consider the following, where Alice and IpswichHospital are suitably defined integers denoting a user and an organisation id.

```
definition Alice_URP :: urp
where Alice_URP =
  (|nhs_id=Alice, org=IpswichHospital, role=ClinicalPractitioner,
   aows={}, activities={ManageSealedDataAccessPermissions}|)
```

Workgroups and Legitimate Relationships

Workgroups are modelled as sets of URPs, and not as sets of users, reflecting the way this is handled in the NHS. We also define a workgroup id used for their identification.

```
type_synonym wg_id = int
type_synonym workgroup = urp set
```

Legitimate Relationships need to contain several kinds of information, namely their type, their status, and of course the information about the two entities of the relationship. The type and status are modelled as a datatype:

```
datatype LR_Type =
PatientReferral | PatientSelfReferral | PatientRegistration |
GPRegistration | SubjectAccessRequest | PatientComplaint |
ExpressedPatientConsent | Legal | SelfClaimed | ColleagueGranted

datatype LR_Status = active | inactive | frozen | expired
```

As an LR is a relationship between a patient and either a workgroup or a single user, we have to model this distinction as well:

```
datatype lr_to = WG wg_id | User urp
```

Finally, together with an identifier modelled as integer, a full LR is a record consisting of the following elements:

```
record LR =
 lr_id :: lr_id
 lr_patient :: patient
 lr_to :: lr_to
 lr_type :: LR_Type
 lr_status :: LR_Status
```

That is, an LR only contains a reference to a workgroup identifier. The information to which concrete workgroup this identifier belongs to is stored separately (as part of the security context introduced below).

Sealing and Consent Information

Each care record contains a flag denoting the consent status. This is modelled as a datatype:

```
datatype consent_flag = opt_out | ask | dontask | suppressed | unknown
```

Similar for the sealing status of an entry, where we have five possibilities:

```
datatype seal = seal_open wg_id | seal_lock wg_id |
                seal_patient | not_sealed | not_sealable
```

Summary Care Records

A Summary Care Record (SCR) essentially consists of three parts:

- General information contained in every record

- Additional Personal Demographic Service (PDS) data
- Medical content (called Personal Spine Information Service (PSIS))

The distinction between the PDS and the PSIS is as follows (taken from [Dep10]):

- **PDS:** The Personal Demographics Service (PDS) is the central and single source for patient demographic information, such as NHS number, name, address and date of birth. It can also contain a much wider range of information to allow for circumstances where a patient may be residing with a relative during recuperation, enabling important correspondence to be sent to the correct address.
- **PSIS:** The Personal Spine Information Service (PSIS) is a central database containing clinical records for each NHS patient. The PSIS record provides an up to date summary of information and key events in a patient's life and care, drug allergies, operations, conditions, medication history as well as details of contacts with care providers.

A PDS entry consists of the following three elements, where the provider is the user who provided this entry:

```
record PDS_entry =
 PDS_id       :: entry_id
 PDS_provider :: user
 PDS_content :: content
```

An entry in the PSIS is similar but additionally contains the sealing status and the type of this entry. Note that a PDS entry cannot be sealed.

```
record entry =
 entry_id      :: entry_id
 entry_type    :: entry_type
 seal          :: seal
 provider      :: user
 entry_content :: content
```

We do usually not care what the actual content of an entry is, as this is not relevant for the access control decision. Thus it can best be modelled as a dummy value, or, if desired, as a string.

A care record is formalised as the following record:

```
record SCR =
 patient_id :: patient
 name       :: name
 address    :: address
 dob        :: dob
```

flag :: consent_flag
GP :: user
deceased :: bool
sensitive :: bool
PDS_ext :: entry_id ⇀ PDS_entry
content :: entry_id ⇀ entry

That is, both the PDS and the proper contents are stored as a partial functions from an identifier to an entry record. The record itself additionally contains information about the name, the address, and date of birth of the patient, his GP, and flags indicating if the patient is deceased or has a sensitive status (in which case special rules would apply). It must be noted that many of these entries are not important in most test purposes. Whenever that is the case, it is best to set them to some dummy value, in order to avoid generating superfluous test cases.

5.2.2 The State

The state of the system, or more exactly, the policy-relevant part of the state of the system, consists of three parts:

1. **Spine.** Modelling the existing care records in the system. Represented as a partial function from patient to SCRs. Note that the term Spine is not used in its standard meaning here.

   type_synonym Spine = patient ⇀SCR

2. **User Context** υ. Storing all attributed URPs. This is needed as a user may only access the system with a URP that belongs to him:

   type_synonym υ= user ⇀(urp set)

3. **Security Context** Σ. Storing the existing workgroups and Legitimate Relationships.

   type_synonym Σ= LRs × WGs

   where

   type_synonym LRs = patient ⇀(lr_id ⇀LR)

   and

   type_synonym WGs = wg_id ⇀workgroup

   We make the assumption that *lr_id*s are unique within the LRs of one patient, and the *wg_id*s are unique globally.

5.2.3 Operations

Next, we model 29 operations, which are generic abstractions of the functional behaviour governed by the Information Governance (IG) principles that may be implemented in applications. All of them are about creating, editing, reading, and deleting an SCR or parts of it, including consent information and seals, or changing or querying the user or security context. Note that these operations are not equal to the activities mentioned earlier. They are part of the model only and restricted to policy-related parts of the system. When performing the tests, these operations need to be mapped to concrete functions. The operations are (informally) grouped into six categories, depending if they deal with complete care records, individual entries, the PDS content, the consent status, LR or workgroup management. At this stage, we merely provide their name and the type of their inputs. Their behaviour is modelled in the policies and in the state transitions.

```
datatype Operation =
(* Operations dealing with care records *)
  createSCR urp patient name address dob consent_flag user
| extendSCR urp patient entry
| readSCR urp patient
| deleteSCR urp patient

(* Operations dealing with the entries of a care record *)
| editEntry urp patient entry_id entry
| removeEntry urp patient entry_id
| readEntry urp patient entry_id

(* Operations dealing with the PDS part of a care record *)
| readPDS urp patient
| editPDS urp patient name address dob consent_flag user bool bool
| extendPDS urp patient PDS_entry
| removePDSEntry urp patient entry_id

(* Operations dealing with the consent flag *)
| changeConsentFlag urp patient consent_flag
| readConsentFlag urp patient

(* Operations dealing with Legitimate Relationships *)
| createLR urp patient lr_to LR_Type LR_Status
| removeLR urp patient lr_id
| getLR urp patient
| changeLRStatus urp lr_id LR_Status
| claimLR urp patient LR
| grantLR urp patient urp
```

```
(* Operations dealing with workgroups *)
| createWG urp wg_id "urp set"
| deleteWG urp wg_id
| addToWG urp wg_id "urp set"
| removeFromWG urp wg_id "urp set"
| addWGLink urp wg_id wg_id
| removeWGLink urp wg_id wg_id

(* Operations dealing with the sealing of entries *)
| createSeal urp patient entry_id seal
| removeSeal urp patient entry_id
| breakSeal urp patient entry_id

(* Operations dealing with URPs *)
| createURP urp user urp
| removeURP urp user urp
```

We do not model changes in the mapping of the functions of an application to the activities, as they happen only exceptionally.

In some cases we might be interested in the output of an operation. We model the possible outputs as an enumeration type, including one element with an arbitrary string and the possibility to concatenate several outputs. Note that for many test purposes, the concrete output is irrelevant and can be set to the `unit` type.

```
datatype Output =
OutEntry entry | OutSCR SCR | OutMsg string | Conc Output Output
```

As an example, the following sequence of operations describes a system execution where first user Bob presenting one of his URPs adds a URP of user John to a specific workgroup, and then John wants to read the SCR of patient Pablo.

```
[(addToWG urp_bob 1 {urp_john}), (readSCR urp_john pablo)]
```

5.2.4 RBAC Policy

In this section, we formalise the Role-based Access Control (RBAC) part of the NPfIT policy. As described in the previous section, the main ingredients of RBAC are URPs, roles, Areas of Work, activities, and functions.

The policy consists essentially of several mappings:

a) A mapping between roles and activities

b) The hierarchy on the activities

c) A mapping between roles and Areas of Work and activities

d) A mapping between an application's functions and activities.

The first three mappings are static and the same for every application, while the last one is application-dependent (but also considered to change only externally). All of them are modelled as simple relations.

In more details, the first three mappings form the National RBAC Database (NRD) and are modelled as a triple of sets as follows:

type_synonym NRD = (Role × Activity) set ×
(Role × AoW × Activity) set ×
(Activity × Activity) set

In addition to that, there is the mapping from the operations to the activities. Unlike the other mappings, this one is part of the model only and has no correspondence in the real systems. Each operation is mapped to a set of activities, where the same activity might be linked to several operations. The semantics employed is that a user is allowed to execute an operation if and only if it maps to at least one activity that the user is entitled to execute.

Depending on the test scenario, we must come up with different kinds of RBAC policies:

- $Function \times user \times urp \times \upsilon \mapsto unit$ when we want to test the correctness of the RBAC implementation of a specific application.
- $Operation \times \upsilon \mapsto unit$ where the operations are mapped to an application's functions, if this policy is combined with other concepts to test the correct IG implementation of a specific application.
- $Operation \times \upsilon \mapsto unit$ where the operations are mapped to activities, if this policy is combined with other concepts to test the IG principles independently from a concrete application.

In all cases, the relevant input state of the policy is the user context υ, which stores the allocated URPs. RBAC neither needs the Spine nor the security context.

Remember that the other policy concepts and the state transitions are only defined over the operations, not the functions or the activities. Thus, when testing a concrete application we need to apply one of the following strategies, corresponding to the above three cases:

- We want to generate directly descriptions of the execution of an application function. In that case, each function must be mapped to one concrete operation in the

model, and this mapping needs to be applied to the inputs when combining the RBAC policy with the other policy concepts and the state transitions.

- We want to generate operations, but they are afterwards directly mapped to a function of the application. That is, we would usually only consider a subset of the operations. This is done with a post-processing step after test case generation taking the mapping into account.
- We want to generate operations only, and have no particular application in mind when generating the test cases. The test engineer (or an automated tool) will map the test data to concrete functions only during test execution.

The first strategy should be employed when testing a new application exhaustively. However, in many cases, especially when focusing on the behaviour related to the Information Governance policies, the second strategy should be preferred as it reduces unwanted complexity.

As an example, assume we have a concrete instance of an NRD, a mapping from the operations to the activities called $op2NRD$, and we want to transform this to a policy of the second kind.

This can be done by the following definition:

```
definition NRD_Policy = OpActivityPolicy op2NRD (mk_ActivityPolicy NRD)
```

where

```
definition mk_ActivityPolicy :: NRD ⇒activityPolicy
where mk_ActivityPolicy nrd =
λ (a, u, urp, uc). case uc u of
  ⊥⇒ ⌊deny ()⌋
| ⌊urps⌋ ⇒ if urp ∉ urps then ⌊deny ()⌋ else
              if  activity_granted a urp nrd
              then ⌊allow ()⌋
              else ⌊deny ()⌋
```

transforms an NRD into a policy with the intended behaviour and

```
definition
OpActivityPolicy op2Act rbac =
λ x. case op2Act (fst x) of
  ⌊fs⌋ ⇒ if (∃f∈fs. (rbac (f, nhs_id (userOfOp (fst x)), (userOfOp (fst x)),snd x)
        = ⌊allow ()⌋)) then ⌊allow ()⌋ else ⌊deny ()⌋
  |⊥ ⇒⊥
```

takes an operation to activity mapping and a policy of type $Activity \times user \times urp \times \upsilon \mapsto unit$ and transforms it to a policy taking as input an operation and a user context which

will be allowed if the operation maps to an activity which is allowed.

The definitions for the other two cases are very similar. In essence, all of them are built up by using the provided relations, a mapping for the operations, and a decision function that implements the RBAC behaviour: a function is granted to a user if he presents a valid URP that allows him to perform an activity that is mapped to the desired function.

5.2.5 Patient's Consent Policy

The concept of Patient's Consent (PC) governs whether a care record can be created and data be uploaded to it. To enforce this, every SCR contains a flag which can take on five different values:

datatype consent_flag = opt_out | ask | dontask |
suppressed | unknown

They have the following meaning:

opt_out: The patient has explicitly chosen not to have a care record. It is not possible to upload medical data, however there is still a record containing demographic information.

ask: The patient wants to have a care record, however users must ask him every time they want to upload any new data.

dontask: The patient wants to have a care record, however he does not want to be asked again before uploading.

suppressed: The patient had an SCR but has chosen to have it deleted. Some information will however be retained and made available for reading for some time for administrative and legal reasons.

unknown: The wish of the patient is unknown. Currently, this is interpreted as *ask*.

The rules about patient consent have the following type:

type_synonym PCPolicy = (Operation × Spine) ↦unit

This policy needs the Spine as state information to the input because the current consent flags are stored as part of the care records. Only a limited number of operations is governed by these rules. They are stored in a set called PC_Relevant_Ops.

Next, we define the individual rules modelling the desired semantics of the consent flag. As an example, the following rule allows all operations if the flag is set to dontask:

```
definition dontaskPolicy :: PCPolicy
where dontaskPolicy = λ(op,sp).
  if    op ∈ PC_Relevant_Ops
  then case SCROp (op,sp) of
         ⌊s⌋ ⇒ (case flag s of dontask ⇒ ⌊allow ()⌋
       | _ ⇒ ⊥)   | _ ⇒ ⊥
  else ⊥
```

Here, SCROp (op,sp) returns ⌊SCR⌋ as specified by the input of the operation op and the Spine sp or ⊥ if it does not exist. The other rules are similar, and the full Patient's Consent (PC) policy is the override (⊕) of all these rules, with the default allow rule for the non-matching inputs.

5.2.6 Legitimate Relationship Policy

Remember that there are ten different types of Legitimate Relationship (LR) and that the LRs also have a status. The policy about LR needs as context information all the existing LRs and the workgroup memberships. These are stored in a security context Σ:

```
type_synonym Σ= (patient ⇀LR set) × (wg_id ⇀workgroup)
```

An LR policy is of the following type:

```
type_synonym LRPolicy = (Operation × Σ) ↦unit
```

Some of the rules of the LR policy are transparent with respect to the type of the LR, others are specific. A function called hasLR returns True if the given user has an *active* LR of any type with a specific patient in the given security context, and self-explaining variants of it (for example hasLR_notSelfClaimed) also exist. A typical rule about the concept of LRs then looks as follows:

```
fun LRPolicy1 where
 LRPolicy1 ((editEntry u p e_i e), Σ) =
                 if    hasLR u p Σ
                 then ⌊allow ()⌋
                 else ⌊deny ()⌋
|LRPolicy1 x = ⊥
```

Other rules (mainly those about how LRs can be transferred) additionally need to take the concrete type and status of an LR into account. As an example, the following rule formalises that a user may only grant an LR if he has an existing LR with the corresponding patient which was not self-claimed:

```
definition grantLRPolicy :: LRPolicy
where grantLRPolicy x = case opLR x of
   grantLR u_from p u_to ⇒
       (({y. hasLR_notSelfClaimed (u_from) p (Σ y)} ◁A_U)
     ⊕({y. ¬ hasLR_notSelfClaimed (u_from) p (Σ y)} ◁D_U)) x
  | _ ⇒ ⊥
```

Note the different format of the last two rules. Both forms are equivalent. The complete LR policy would consist of a very large number of rules of these two kinds. Again, all these rules can be combined straightforwardly using the override operator with a default allow rule in the end.

5.2.7 Sealed Envelopes Policy

Each content entry of an SCR has a flag showing the sealing status of that entry. The flag can take on any of five values:

seal_open wg: The entry is sealed with an open seal. Only users in the workgroup with id wg can read this entry, but others may know that the entry exists, and override the seal when this is justified.

seal_lock wg: The entry is sealed with a locked seal. Only users in the workgroup with id wg can read this entry or know that the entry exists.

seal_patient: The entry is hidden from the patient.

not_sealed: The entry is not sealed.

not_sealable: The entry must never be sealed.

The rules about SE use information from the care records and the workgroup memberships, which are stored in the security context Σ. Thus, their type is:

```
type_synonym SEPolicy = (Operation × Spine × Σ) ↦unit
```

A user is allowed to read an entry directly (i.e. without breaking a seal), if the entry is either not sealed or, else, he is a member of the respective workgroup. Such a rule can be modelled as follows, where userHasAccess checks membership in an allowed workgroup if required:

```
definition readEntry :: SEPolicy
where readEntry x = case x of
  readEntry u p e_id,S,(lrs,wgs) ⇒
              case get_entry S p e_id of
                   ⊥   ⇒ ⊥
```

```
                | ⌊e⌋ ⇒ if (userHasAccess u wgs e)
                          then ⌊allow ()⌋
                          else ⌊deny ()⌋
| x ⇒ ⊥
```

The other rules follow the same pattern and can be combined as usual using the override operator. We must, however, not forget rules specifying that only a seal_open seal can be broken and that a not_sealable seal must never be sealed.

5.2.8 State Transitions

As we want to test a system that changes over time, we need to model state transitions. The relevant state in this case consists of three individual parts: the Spine, the security context, and the user context. The state transitions are triggered by an operation and there are always two cases: a transition if the operation is allowed by the policy, and a transition if the operation is denied by the policy. We model each of these state transitions individually, thus leading to six different transitions. In the following, we show an excerpt of the state transition for an allowed operation on the Spine.

```
fun Spine_A :: (Operation × Spine) ⇀Spine where
 Spine_A ((extendSCR u p e), S) =
 case S p of ⊥⇒ ⌊S⌋
           | ⌊x⌋ ⇒ ⌊(S(p↦x(|content := (content x)
               ((SOME y. y ∉(dom (content x)))↦e)|)))⌋
```

This is the state transition for the Spine part of the state that should be applied when the extendSCR operation is allowed to execute. The state will not change if there is no care record available for that patient. Otherwise, the content part of his or her care record will be extended with a new entry, where the id of the entry is some yet unused one.

Note that most operations only modify one part of the state, and leave the other two intact. Additionally there are operations that do not modify the state at all (the read operations).

Note that in the above case, the state was not transformed in case of invalid input parameters (for example when the patient does not exist). This means that the policy could still return an allow, but the operation will just be a noop. Depending on the concrete behaviour of the system under test, an alternative approach might be preferred: there is additionally a “functional policy”, which only allows the operations if all the inputs are well-defined. Furthermore, one could model the output messages to output an error message in case of ill-defined inputs. There is no single solution which will be good for all cases; the modelling strategy depends both on the system under test and on what we want to test.

While not state transitions technically, *outputs* are modelled in a very similar way, which is why we also introduce them here. They are formalised using a partial function having an operation and the relevant parts of the state as input, and the Output type as output. In more detail, this function has the following type in our model:

Operation × Spine ⇀Output

Again, we model the output in two ways, depending if the policy allows or denies the corresponding operation. As an example, the output for the *readSCR* operation is formalised as follows:

fun
output$_A$ ((readSCR u p), S) = case S p **of** ⌊x⌋ ⇒⌊OutSCR x⌋
| ⊥ ⇒ ⌊OutBlank⌋

That is, if the corresponding care record exists in the current state, it is output, otherwise the message "OutBlank" denoting no output is used.

The individual state transitions and the functions defining the output can be combined to a single transition system using the standard UPF operators. The following, for example, is a model of the system behaviour if all operations would be accepted, simulating the behaviour of the system if there was no policy:

definition A$_{ST}$ :: Operation × Spine × Σ× υ⇀Output × Spine × Σ× υ
where
A$_{ST}$ = output$_A$ ⊗$_M$ Spine$_A$ ⊗$_S$ Σ$_A$ ⊗$_S$ υ_A o (λ (a,b,c,d). ((a,b),(a,b,c,d)))

D$_{ST}$ is defined similarly, but in most cases does not change the state.

5.2.9 Combination

So far, we have modelled only small individual parts of the system and its policy. In the end, all of them have to be combined to a transition policy to model the desired real behaviour. First, all the policy parts can be combined:

definition appPolicy :: (Operation × Spine × Σ× υ) ↦unit
where appPolicy = C$_1$ o$_f$ ((C$_1$ o$_f$ ((C$_1$ o$_f$
(PCPolicy
⊗$_{\vee D}$ SEPolicy) o C$_2$)
⊗$_{\vee D}$ LRPolicy) o C$_3$)
⊗$_{\vee D}$ AppRBACPolicy) o C$_4$

In this example, we use the application-independent RBAC policy. The definition would slightly change if taking an RBAC policy including application functions. The coercion functions

- $C_1 = \lambda(a,b).\ a$
- $C_2 = \lambda(a,b,c).\ ((a,b),(a,b,c))$
- $C_3 = \lambda(a,b,c).\ ((a,b,c),a,c)$
- $C_4 = \lambda(a,b,c,d).\ ((a,b,c),(a,d))$

serve to the usual technical *repackaging* of the underlying state and input formats involved in the composition. This policy can then be combined with the previously combined two state transitions as follows, where $C_5 = \lambda a.\ (a,a)$:

definition policy :: $(\text{Operation} \times \text{Spine} \times \Sigma \times \upsilon) \mapsto (\text{Output} \times \text{Spine} \times \Sigma \times \upsilon)$
where policy = $C_1\ o_f\ (A_{ST}, D_{ST}) \otimes_\nabla$ appPolicy o C_5

In the end, the complete policy, together with the description of the system behaviour and the output of the applications, is completely stored in the transition policy *policy*. Note that this policy will be different for most systems under test, because they usually need to implement a different set of rules and also the state transitions and the output messages do not necessarily have to be the same among different applications.

5.2.10 Integrating Logging

One thing which is one the one hand rather special with regards to general security policies, but on the other hand very common in the health-care domain is that non-legitimate accesses should still be possible in exceptional circumstances. It would therefore be valuable to check if the UPF could be adjusted to support such break-glass policies, where users should be able to override the policy decisions in some circumstances. Full support for break-glass is however not directly needed in the NPfIT policies, which use a simpler concept.

In the NPfIT, applications are supposed to log all policy overrides such that a privacy officer might inspect them. As this is a very important concept regarding confidentiality, support for testing for compliance to the logging requirements is beneficial. Our previously presented model can be extended to support this as follows:

- The two operations `breakSeal` and `claimLR` already model the two ways of policy override to be supported.
- As additional part of the system state, we model a log, i.e. a sequence of messages. Each time such an operation was triggered, a new message containing the required information is appended to the log state.

5.3 Test Case Generation for NPfIT Applications

In this section, we briefly present how to generate test data for the policies we modelled in the previous section. The general techniques and methods are the usual ones used in the Unified Policy Framework and are presented in more detail in the firewall case study. Here we outline their applicability to the NPfIT context. We differentiate between unit tests, denoting a single access to an application together with the expected outcome, and sequence tests, denoting a sequence thereof.

5.3.1 Testing Strategy

Policy conformance testing of applications of the NPfIT must be integrated into the existing testing process to be most valuable. Furthermore, testing of conformance to the IG policies of the NPfIT is domain-specific. The employed strategy should be chosen depending on what other kinds of tests are being executed, how much the application has already been in use, where it will be applied, etc.

In the following, we outline our ideas on one possible testing scenario for the following use case: a supplier that delivers one or more applications to one or more NHS sites needs to ensure conformance of the application to the Information Governance Policies. The supplier is not necessarily the programmer of the applications and might not even have access to the source code. The code-base of the application is expected to already have passed the basic tests and supposed to have reached a certain correctness criteria already.

The supplier wants to validate the following not independent aspects:

1. Does the application implement the policy concepts correctly?
2. Does the application implement the correct version of the specification of the policy concepts?
3. Does the application apply a policy configuration correctly?
4. Are the applications, once delivered, configured correctly?

Points 1-3 and parts of the last one will not be tested using live data. It can be expected that in such a big project, there will special testing environments available where applications can be tested using artificial data. Testing a live system would require more precautions, which we do not consider part of this work.

The first step towards such a test scenario is to create the concrete model. This step was outlined in the previous section. Next, the conformance could be tested using the following three strategies:

- Apply thorough unit tests. Due to the size of policy, there will not be single, rather unconstrained tests, but a large number of more specific ones. Especially there will be tests testing the individual policy concepts only.
- Create a number of sequence tests, by providing test specification automata specifying "interesting" tests.
- Combine the last two points. That is, drive the system into interesting states using sequence testing and apply thorough unit testing in this states.

We will elaborate on the unit and sequence tests in the following. There is always the option of applying the tests in an empty state. However, this would not lead to interesting tests. Better suited is to start the tests in a starting state provided in the test specification, already containing a couple of user, patients, care records, etc. Note that part of this starting state might be unknown.

5.3.2 An Example Scenario

First, we define an example scenario, serving as the starting state of the test data to be generated in the sequel. It consists of the following setting:

- Three users: Alice, Bob, and John. Alice is attributed two URPs, one in the role of a Clinical Practitioner, the other as a Clerical (an administrative role). Bob is a nurse, and John as well. There are no areas of work in the URPs, but a few additional activities.
- Two workgroups: the first with the first URP of Alice and John, the second with Alice's other URP and Bob.
- Two patients, called patient1 and patient2.
- Two Legitimate Relationships, one between patient1 and workgroup 1, the other between patient2 and workgroup 2, both of type GPRegistration
- Two care records, one for each patient.

In the following, we show the definitions of the entries, the care records, the Spine, and the starting state σ_0 (note that we use a slightly simplified representation):

definition entry1 :: entry
where entry1 = (1, medication, seal_open 1, alice,data)

definition entry2 :: entry
where entry2 = (2, medication, seal_open 2, alice,data)

definition entry3 :: entry
where entry3 = (3, medication, not_sealed , alice,data)

definition SCR1 :: SCR
where SCR1 = (patient1, ''patient1 '', '' london '',
14, dontask, alice , False, False,
$\emptyset$, $\emptyset(1\mapsto\text{entry1})(2\mapsto\text{entry2})(3\mapsto\text{entry3})$)

definition SCR2 :: SCR
where SCR2 = (patient2, ''patient2 '', '' ipswich '',
14, dontask, 14, False, False,
$\emptyset$, $\emptyset$)

definition Spine_0 :: Spine
where $\text{Spine}_0 = \emptyset(\text{patient1}\mapsto\text{SCR1})(\text{patient2}\mapsto\text{SCR2})$

definition σ_0 :: Spine $\times\Sigma\times\upsilon$
where $\sigma_0 = (\text{Spine}_0,\Sigma_0,\text{UC}_0)$

5.3.3 Unit Tests for the NPfIT Policies

Unit tests allow for a deep exploration of the (static) policy space. As such, we will need to generate a relatively large amount of test cases ensuring that all parts of the large and complex policy are tested.

The first category of tests will be used for testing each of the individual policy concept on its own. Consequently the policy model to be used for test case generation will not be the full combined policy, but rather only the individual parts of it.

One example of such a test specification is:

test_spec $\bigwedge$x u p. x = readSCR u p $\Longrightarrow$
SUT x = LR_Policy (x,Σ_0)

This test specification will lead to test cases of a read access to a care record, checking that the access will only be granted if the user is in a workgroup having a legitimate relationship with the patient.

A refinement of the last test specification is to ensure only test cases of valid users (i.e. there is a user which executes the operation with one of his valid URPs stored in the user context) and patients (i.e. a patient for which there exists a care record) are being created:

test_spec ⋀x u p.
⟦p ∈ dom(Spine$_0$); existingURP u UC$_0$; x = readSCR u p⟧ ⟹
SUT x = LR_Policy (x,Σ_0)

The generated test cases are similar in both cases, however in the latter case they will be more tailored to the concrete system state while in the former case more invalid data would be created. Such rather constrained test cases will be used to test against specific parts of the specification, as for example things which are explicitly stated in the informal specification documents.

Other test specifications will be less constrained. As an example:

test_spec ⋀x. SUT x = LR_Policy (x,Σ_0)

will generate 94 test cases, using all operations available.

Similar test specifications can be created for testing conformance to the concepts of RBAC, Sealed Envelopes, and Patient Consent.

As an example, the following policy combines three of the four concepts and generates test cases when a user wants to read a valid and existing entry of the user. Remember that this is only possible if the user has an existing LR with that patient, the entry is not sealed from him, and the patient has consented for the care record to be shared:

test_spec ⋀x p u e.
⟦existingEntry p e Spine$_0$; existingURP u UC$_0$; x = readEntry u p e⟧ ⟹
SUT x = SE_LR_PC_Policy (x, Spine$_0$, Σ_0)

The above examples had as outcome unit only. Depending on the test purpose, the concrete values of the outcomes also have to be tested.

Other test scenarios are used for testing critical situations. For example, we might want to validate that the personal General Practitioner (GP) of a patient is always allowed to read his Summary Care Record (SCR).

A policy specified in a wide range of formal and informal documents, and guidelines is prone to be underspecified or to contain ambiguities. In the case of the NPfIT this has already been observed before [Bec07]. In such known cases, where a policy specification can be interpreted in several ways, we can create tests to check to which interpretation a specific application conforms to.

For example an early ambiguity detected by [Bec07] is whether users are able to seal data they are not allowed to read. We can test this property as follows:

⟦Policy ((readEntry u p e), σ_0) = ⌊deny ()⌋⟧ ⟹
SUT (createSeal u p e s,σ_0)= ⌊deny ()⌋

5.3.4 Sequence Tests for the NPfIT Policies

Given the inherently stateful behaviour of the NPfIT policies, unit testing can only validate the systems to a certain extent, and sequence tests play an important role. Furthermore, generating test cases that ensure the compliance to standards and regulations like the principles from the Caldicott Report [The97] or stipulations such as the requirement that test results can only be accessed after breaking a seal usually require test specifications for sequence tests.

The sequence testing concepts presented in section 4.3 can directly be applied to the NPfIT scenarios as well.

The general format of a test specification is the following:

P is $\Longrightarrow$
$\sigma_0 \models$(os $\leftarrow$mbind is appMon; (return (os = X))) $\longrightarrow$
SUT (σ_0,is) = X

With the following meaning: Given an input sequence is for which P is true. If the modelled policy monad returns the output sequence X if started in state σ_0, then the system under test is also supposed to return the output sequence X on the inputs is.

The returned test data are sequences of access to a system, together with the expected outcome sequence. An example is the following sequence of length two:

SUT (σ_0, [readSCR urp1_alice patient2, readSCR urp1_alice patient1]) =
[⌊allow (OutSCR SCR1)⌋, ⌊deny OutBlank⌋]

Recalling the observations about achieved specification coverage already give us an indication that the test specification automata in this scenario necessarily have to be constrained.

As an example, consider the following test specification

test_spec
users is $\subseteq$ {urp1_alice, urp2_alice, urp_john, urp_bob} $\Longrightarrow$
$\sigma_0 \models$os $\leftarrow$mbind is appMon; return (os = X) $\longrightarrow$
SUT (σ_0, is) = X

Here, only the users Alice, Bob and John are allowed to perform an operation.

This test specification produces test data like the following:

SUT (σ_0, [readSCR urp_john pablo,
addToWG urp_bob 1 {urp_john},
readSCR urp_john patient1]) =
[deny OutNo, allow OutSuccess, allow (OutSCR SCR1)]

specifying the output that the SUT must produce when receiving the three operations in sequence. Here, John is first denied access to patient1's SCR, but is later allowed after Bob has added him to a workgroup.

Limiting the possible inputs allows for limiting the adversary to special kinds of operations. As an example, we might make the assumption that an attacker can only access the system using a valid URP, reflecting measurements in place which are not part of our models (for example access is only possible using legitimate smart cards).

5.3.5 Test Execution for NPfIT Policies

The concepts presented in section 4.4 for executing the tests can also be applied here.

One straightforward approach to executing the generated tests is by integrating execution of the test data generated by our framework into the traditional testing approach based on standard testing tools. The test data exporting facility of HOL-TestGen can be adapted to be used for import into such a tool, if this tool does actually provide some kind of import facility.

Additionally, the exporting facility can be adapted to output a human-readable detailed script that a test engineer can use to apply the tests on a real system. For example, the following verbose script of a sequence of length three could be generated:

- User Charlie, in his role as Clerical, adds a new care record to a patient identified as John.
- This operation should succeed.
- Then, Charlie in still the same role Clerical, adds a new Legitimate Relationship, with the identifier "4", between the patient John and the user Bob.
- This operation should succeed.
- Then, log in as user Bob in his role as ClinicalPractitioner. Bob then deletes the care record of patient John.
- This operation should fail.

While this approach is relatively easy to implement, the value is also smaller than what would be possible in the framework. In general, one of the advantages of model-based testing over other testing approaches is the possibility of automating large parts of the testing process. For the case where the application to be tested has a standardised interface based on WSDL-based web services, we have already presented a solution in section 4.4. While we did not apply this approach to a real application of the NPfIT, we have developed a small prototype serving as a proof of concept on which we have executed the tests.

5.4 Summary

In this chapter, we have presented the instantiation of the Unified Policy Framework (UPF) with the domain of the Information Governance Policies of the National Programme for Information Technology (NPfIT). This case study serves to provide evidence of the quality of the Unified Policy Framework as the IG policies are an example of a complex security policy system consisting of a number of distinct concepts having to be integrated. Furthermore, these concepts do not follow any existing formal access control model, and thus require a very flexible modelling framework. We further outlined how to apply the testing methods of the framework to this domain. The test data can be used for automatic test execution on those applications which provide their interface using WSDL-compliant web services.

6 Model-based Firewall Conformance Testing

In this chapter, we present a large-scale instantiation of the Unified Policy Framework in the domain of firewalls and networks. We will first provide a short introduction into the domain context and proceed with presenting how these concepts can be modelled within the UPF. This is followed by a thorough presentation and evaluation of model-based conformance testing of such systems, for which the firewall and network domain is very well suited due to the fact that while integrating almost all central concepts of both the UPF and of security policies in general, it remains rather simple conceptually.

6.1 Firewalls and Networks

It is common knowledge that unrestricted access to the Internet is a security risk. Firewalls are active network elements that can filter network traffic. They are a widely used tool for controlling the access to computers in a network and services implemented on them. In particular, firewalls filter, based on different criteria, undesired traffic out of the data-flow going to and coming from a network. Their intended behaviour as specified in the *firewall policy*, varies from network to network according to the needs of its users. Therefore, firewalls can be configured to implement various security policies.

Firewalls suffer from the same quality problems as other complex software, but mature products from established and trusted vendors are often considered trustworthy and any vulnerabilities are often believed to be found and patched relatively quickly. Given this situation, is there need for anyone other than firewall manufacturers, independent labs, and organisations with critical security requirements to test firewalls? The answer is "yes," because the off-the-shelf product must be configured with a rule set that implements an appropriate security policy to create a working firewall. The likelihood of a security vulnerability arising from misconfiguration is much greater than from a bug in the software itself.

This is especially true for firewalls, as they tend to have complicated configurations. Furthermore, firewall policies and hence rule sets change with time. Often, changes are implemented by adding rules, resulting in ever-growing complexity, which increases the probability of errors and the challenge of finding them. Thus, there is a continuing

need to re-validate that the configured firewall complies with the security policy, and the importance and difficulty of this increases with time. While it is useful to verify the rule set by inspection helped by such analysis tools as are available, this is no substitute for actually testing that the firewall really behaves as specified by the security policies from which the rule set is derived. Furthermore, firewall vendors tend to update their products regularly. A firewall user needs some kind of testing mechanisms to ensure no bugs are being introduced silently.

Among others, the following scenarios might be envisaged using a model-based firewall conformance testing approach:

- Testing for policy compliance of a single network component (firewall, router)
- Testing for implementation correctness of a network component
- Testing that a set of connected network components conform to a high-level security policy
- Testing for connectivity in a network

In other words, we do not necessarily have to restrict ourselves to one single device to be tested against. Rather, there could be several of them in an arbitrary topology. From a testing point of view, only the endpoints of the network are relevant.

Next, we introduce the firewall domain in more detail. A *message* that should be sent from network A to network B is split into several *packets*. A packet contains parts of the content of the message and routing information. The routing information of a packet mainly contains its source and its destination address.

A *stateless firewall*, also called stateless packet filter, filters traffic from one network to another based on certain criteria as for example the source and destination address of a network packet. The *policy* is the specification of the firewall which describes which packets should be denied and which should be accepted.

Table 6.1 shows a firewall policy as it can be found in security textbooks for a common firewall setup separating three networks: the external Internet, the internal network that has to be protected (intranet), and an intermediate network, called the demilitarized zone (DMZ). The DMZ is usually used for servers that should be accessible both from the outside (Internet) and from the internal network (intranet) and thus are governed by a more relaxed policy than the intranet.

Such a table-based presentation of a policy uses a first-fit matching strategy, i.e. the first match overrides later ones. For example, a packet from the Internet to the intranet is rejected (it only matches the last line of the table), whereas an smtp-packet from the intranet to the DMZ is accepted. Lines in such a table are also called *rules*; together, they describe the *policy* of a firewall. As usual, we will use the terms policy and rules

source	destination	protocol	port	action
Internet	DMZ	smtp	25	accept
Internet	DMZ	http	80	accept
intranet	DMZ	smtp	25	accept
DMZ	intranet	smtp	25	accept
intranet	DMZ	imaps	993	accept
intranet	Internet	http	80	accept
any	any	any	any	deny

Table 6.1: A simple firewall policy separating three networks. Such a policy table uses a first-fit pattern matching strategy, i.e. the first match overrides later ones. For example, a packet from the Internet to the intranet is rejected (it only matches the last line of the table) whereas a http-packet from the intranet to the DMZ is accepted.

synonymously.

A firewall might discriminate on further fields of a packet. A notable and often seen example is the indication of the network protocol, i.e. TCP, UDP, or ICMP. Furthermore, a firewall often performs additional duties next to filtering of packets. In this chapter, we will also introduce a technique to model and test address translation behaviour as encountered for example in Network Address Translation (NAT).

Furthermore, the stateless scenario just introduced does not solve all security problems of modern network infrastructures. There are several network protocols that require the firewall to observe the internal state of a protocol run, which therefore requires a *stateful firewall*. A stateful firewall keeps track of the existing connections and can adapt its filtering behaviour based on this. As an example, the well-known File Transfer Protocol (FTP) is based on a dynamic negotiation of a port number which is then used as channel to communicate the file content between the sender and the receiver. Thus, a stateless firewall can only provide a very limited form of network protection if protocols such as FTP are involved, whereas a stateful firewall that observes the inner state of the FTP session can open the negotiated port dynamically. Similarly, modern Voice over IP (VoIP) protocols as well as protocols used for streaming multimedia data use dynamically negotiated protocol parameters for optimising the network load.

6.2 Modelling Firewall Policies

In this section, we present a large-scale instantiation of the UPF with the domain of firewalls and networks, as described in the previous section. We first give an overview of how to model the constituent parts of the domain: networks, addresses, and network

packets. We then model three distinct kinds of policies in that domain: stateless packet filtering policies, network address translation policies, and stateful firewall policies, including their combination.

6.2.1 A Model of Networks

A stateless firewall takes as input a single network packet, and statically decides what should be done with it, independently of what has happened before. We thus need to model all the information that firewalls base their decision on.

Network Addresses

A network address specifies the origin and the destination of a packet. Depending on the test scenario one has in mind, different kind of address representations might be desired. While at times it might be enough to specify the subnetwork a packet originates from or is being routed to, at other times specific host addresses including port numbers are required. For this reason, we allow an arbitrary address representation and merely declare the unconstrained type class α adr. The use of a type class allows us to formalise most aspects of the network communication without relying on a concrete representation of network addresses.

Typically, an address in our terminology consists of a specification of the host or subnetwork, and optionally a port number and a protocol indication, which can be combined to tuples.

In the following, we provide some example address representations. First, three possible host/subnetwork representations, which can arbitrarily be combined with an example of a port number and protocol formalisation:

- *Datatype:* Many policies do not differentiate between the different hosts within a network. Thus, instead of modelling single addresses, we may only specify the network a host is part of by modelling all the relevant networks as an element of a fixed datatype, as in:

  datatype networks = internet | intranet | dmz

  The advantage of this representation is its simplicity; if the model is simple and rather abstract, test case generation is much more efficient. However, the code coverage is possibly also smaller. An additional post-processing step after test data generation can be used to randomly choose a specific host within a network.

  The modelling possibilities, however, are also limited: we cannot distinguish several hosts within a network and the networks must not be overlapping. Thus this model

can only be used in simple scenarios.

- *IPv4:* Another address format tries to stay as close to reality as possible: Internet Protocol (IP) addresses (version 4) are directly modelled as a four-tuple of integers.

  type_synonym ip = int × int × int × int

  The expressiveness of a firewall policy model is significantly extended when using this address representation: each host is specified directly, we can model overlapping networks, and can also directly specify subnet masks. The expressiveness comes with a price however, as also the test case generation complexity is larger. Instead of one variable for each host, there are four. This makes test case generation much slower as the test space is enlarged significantly.

- *Integer:* There is a nice compromise between the last two models: modelling an address as a single integer:

  type_synonym ip' = int

  This model is as expressive as the IPv4 model: a four-tuple of integers can always be transformed into a single integer and vice-versa (after all, an IPv4 address is simply a 32-Bit integer). Compared to the IPv4 model, the state space is reduced significantly, which allows for a deeper exploration of the test space.

These subnet representations can be combined with the concept of source ports, destination ports, and protocols, i.e. depending of the purpose of the model, an address can be modelled as an element of a subnet or it can be refined with either a pair of source and destination ports, a protocol, or both. These concepts can be modelled as follows:

- *Ports:* A common requirement for firewall policy models are port numbers. They can be modelled as integers.

  type_synonym port = int

- *Protocols:* If we need to model a policy that discriminates packets according to the transport layer protocol, we can also model those as part of an address. Usually the number of such protocols is quite small, thus it is best modelled as a datatype:

  datatype protocol = tcp | udp

  At first sight, it might be surprising to model the protocol as part of an address. However, this choice allows to keep the generic model as clean and simple as possible.

For the sake of simplicity we will focus on one address representation throughout the rest of this thesis: an address being a pair of two integers. The first denoting a host (encoding its IP address), the second one the port number. Of course, the formalisations and tests work for other representations equally well.

type_synonym address = int

type_synonym port = int

type_synonym IntPort = address × port

For the sake of simplicity, we require the same format for the source and destination address of a packet. This means that for the protocol field, which needs to be the same for both, only one should be considered and the other one ignored.

Networks

A network is essentially a set of addresses. To allow some further modelling possibilities, we define them as sets of sets of addresses - making it simpler to define some subnetworks forming together one bigger network. In other words, a network consists of a set of subnetworks, and a subnetwork consists of a set of addresses:

type_synonym α net = (α::adr) set set

For checking whether a given address is part of a network, we define the following operator:

definition _ ⊏ _ :: α adr ⇒α net ⇒bool
where a ⊏ S = ∃s∈S. a ∈ S

Packets

Next to a packet's origin and destination, little additional information needs to be encoded into a packet. We allow an identifier (as an integer), and a generic content. Both are used when modelling stateful firewall policies and can be set to some default dummy value when restricting testing to stateless firewalls. Hence, a packet is defined as:

type_synonym (α,β) packet = id × α src × α dest × β content

Further, we define projectors, for example `getId`, for accessing the components of a packet.

6.2.2 A Model of Stateless Firewall Policies

A stateless packet filtering rule takes as input a network packet and either accepts or denies this packet. There is no address translation or a similar action performed. Thus, the type of such a rule is:

type_synonym (α, β) FWPolicy = (α, β) packet $\mapsto$unit

Policy rules are defined as domain restrictions of the elementary policies, as presented in section 3.2. We define a range of combinators that can later be used for specifying concrete policies. While some of these combinators are applicable to all kinds of packets, others are only applicable when using more specific address representations, for example only those containing port numbers.

An example of a typical combinator for such a policy is the following one, which can be used to model a rule that denies all the traffic between two networks. It is defined by restricting the deny-everything combinator to those packets having their origin and destination in the given networks:

definition
deny_from_to :: α::adr net $\Rightarrow\alpha$ net $\Rightarrow$((α,β) packet $\mapsto$unit)
where deny_from_to src_net dest_net =
  {pa. getSrc pa $\sqsubset$src $\wedge$ getDest pa $\sqsubset$dest} $\triangleleft$ D_U

Extended combinators can then be defined by further restricting these combinators, as for example the following one denying a specific port between two networks:

definition deny_from_to_port
where deny_from_to_port src dest port =
  {pa. getDestPort pa = port} $\triangleleft$ deny_from_to src dest

These combinators can then be used to formalise a rule like:

deny_from_to_port intranet internet 14

A stateless packet filtering policy usually consists of an often very large number of such rules which are processed in a first-fit manner and a default rule being used when no other rule matches. For achieving this behaviour, we may use the standard override operator $\oplus$.

The filtering policy of the example introduced in section 6.1 is defined as follows. This specification is very similar to how it is configured in common firewall tools.

definition Policy =
  allow_port_from_to intranet internet 80 $\oplus$
  allow_port_from_to intranet dmz 993 $\oplus$
  allow_port_from_to dmz intranet 25 $\oplus$
  allow_port_from_to intranet dmz 25 $\oplus$
  allow_port_from_to internet dmz 80 $\oplus$
  allow_port_from_to internet dmz 25 $\oplus$
  D_U

6.2.3 Network Address Translation Policies

Common firewalls often perform some kind of Network Address Translation (NAT), where either the source or the destination address of a packet is transformed when passing the firewall. There is a wide range of possible transformations, for example:

- all IP addresses in a given range are transformed to a static one,
- all IP addresses in a given range are transformed to an address out of a pool of addresses, or
- all IP addresses in a given range are transformed, as well as their ports. This is called Port Address Translation (PAT).

For these kinds of policies, the outcome must be a description of the transformed network packet. As the firewall can choose between a set of addresses, there is a number of acceptable transformations and the outcome can be modelled as a set of packets. All members of this set denote an admissible behaviour of the firewall. Thus the policy type for these kinds of policies is:

type_synonym NAT_Policy = (α, β) packet $\mapsto$((α,β) packet set)

The decision values of these policies is usually always allow. The rules can be specified by using pre-defined packet transformation functions, mapping them to an acceptance policy, and then applying domain restriction to those packets that are supposed to be transformed. A typical example is a NAT rule allowing one to specify source address transformations to a pool of addresses. The three just mentioned concepts can be defined as follows:

definition src2pool :: α set $\Rightarrow$(α,β) packet $\Rightarrow$(α,β) packet set
where src2pool t = λp.
{(i,s,d,da). (i = getId p $\wedge$ s $\in$ t $\wedge$ d = getDest p $\wedge$ da = getContent p)}

definition src2poolAP
where src2poolAP t = A_f (src2pool t)

definition nat2pool :: α set $\Rightarrow$ α set $\Rightarrow$(α,β) packet $\mapsto$(α,β) packet set
where nat2pool srcs transl = {x. getSrc x $\in$ srcs} $\triangleleft$ src2poolAP transl

The first definition is a function rewriting the source address of a packet, the second one transforms this function to an acceptance policy, and the last one is the provided combinator, restricting this transformation to the relevant input packets. This combinator can be used to define concrete NAT rules.

To get a complete NAT policy, all such rules can be combined using an override operator. Usually a default rule in the end is mapping all packets to a set containing the identity

element only.

Note that the just presented technique of defining NAT rules is general and may be re-used whenever modelling policy rules transforming parts of the input.

6.2.4 Combining Stateless Packet Filter and NAT Policies

A real firewall usually performs several activities in parallel, for example filtering and network address translations. We can use the UPF combinators to combine a stateless packet filtering policy with a policy governing network address translation. Let us assume the stateless packet filtering policies have the following type:

type_synonym Filter = (IntPort,β) packet $\mapsto$unit

and the network address translation policy has the type:

type_synonym NAT = (IntPort,β) packet $\mapsto$((IntPort,β) packet) set

Their combination is:

definition Policy :: (IntPort,β) packet $\mapsto$((IntPort,β) packet set)
where Policy = (λ (x,y). x) o_f (NAT $\otimes_2$ Filter o (λx. (x,x)))

The first coercion function throws away the return value of the second policy (which is just unit), the second one assures that instead of a pair of packets for the input they are identified.

If one considers the standard Linux firewall *iptables*, then in reality the two policies are actually not processed in parallel but sequentially. This behaviour is even more evident when considering a network consisting of a number of network devices all performing policy decisions. For example, a NAT rule might be applied to filtered packets only, or vice-versa. As another example, several filtering policies can be executed sequentially. This behaviour can be specified using the sequential composition operators, leading to an alternative way to combine the two policies in the example above.

The natural disadvantage when using sequential operators is that the types of the constituent policies need to match somehow. As an example, if we apply above NAT policy after the filtered policy, we first need to transform the filtering policy to the type:

type_synonym Filter2 = (IntPort,β) packet $\mapsto$(IntPort,β) packet

where the outcome of applying the policy to a packet is the identity packet (denoting that the packet is "leaving" the firewall unchanged). The two policies can then be composed by:

definition Policy = NAT $\circ_2$ Filter2

There are other cases where both composition possibilities might be used.

6.2.5 Stateful Firewall Policies

Next, we present our model of stateful firewall policies. The first question to be addressed is how to specify the state in question. It clearly needs to contain the currently active policy (which now can change dynamically), as well as some form of a history about the received packets to keep track of the state of the connection. Thus, we define:

type_synonym (α,β,γ) FWState = α × ((β,γ) packet ↦unit)

where a straightforward way to define the history α is to model it as the list of so far accepted packets:

type_synonym (β,γ) history = (β,γ) packet list

The state transitions take as inputs a network packet. Following the standard UPF approach, the state transitions are modelled as partial functions.

type_synonym (α,β,γ) FWStateTransition =
 ((β,γ) packet × (α,β,γ) FWState) ⇀(α,β,γ) FWState

The following two auxiliary functions facilitate defining the concrete state transition as they allow reasoning over the order of elements in a list:

fun before :: (α ⇒bool) ⇒ (α ⇒bool) ⇒ α list ⇒ bool **where**
 before p q [] = False
|before p q (a # S) = q a ∨(p a ∧ (before p q S))

and

fun not_before :: (α ⇒bool) ⇒ (α ⇒bool) ⇒ α list ⇒ bool **where**
 not_before p q [] = False
|not_before p q (a # S) = q a ∨(¬ (p a) ∧ (not_before p q S))

Modelling FTP

At this point, we outline how we model the stateful File Transfer Protocol (FTP), as a typical example of stateful network protocols. In this section, we restrict ourselves to a fixed address model.

During an FTP session, the server opens a data connection to the client using a port that was indicated by the client. Figure 6.1 shows an abstract trace of an FTP session: the client initialises the session by sending a `init` message to the server, next the client

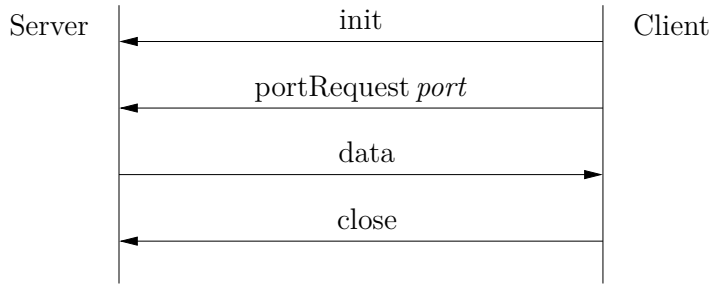

Figure 6.1: An abstract trace of an FTP session

sends a `port request` containing a dynamic `port` for the data connection and then the server sends the `data` to the client using this dynamic `port`. Eventually, the client will `close` the connection and the firewall must close the data port. The messages are identified using the content part of a packet. Thus, the following datatype is used:

datatype ftp_msg =
ftp_init | ftp_port_request port | ftp_data | ftp_close | other

Further, we will use the id part of a packet to distinguish several FTP sessions.

Next, we model the two state transitions. The case for for denial state transition is trivial: the state is not changed:

fun D_{ST} :: ((IntPort,ftp_msg) history, IntPort, ftp_msg) FWStateTransition **where**
D_{ST} (p,s) = ⌊s⌋

For the acceptance state transition, three cases must be distinguished.

1. If a legitimate port request message arrives, the policy is extended by a rule that opens the given port, and the packet is appended to the history.

2. If a legitimate close message arrives, the port is closed. Note that using this semantics, the policy after execution of an FTP protocol exchange does not necessarily have to be the same as it was before in those cases where the original policy allows traffic on this port. If this is not desired, alternative versions of the state transition can be used, either one where the original policy is stored as part of the state and reenacted after the close message, or one where a dedicated status boolean variable stores the information if the port was open at the beginning (the latter is desirable when considering interleavings of protocols). In all cases, the packet is appended to the history.

3. All other messages do not change the policy, but the packets are appended to the history.

Those three cases can be modelled as follows:

```
fun A_ST :: ((IntPort,ftp_msg) history, IntPort, ftp_msg) FWStateTransition
where

(* FTP_PORT_REQUEST *)
 A_ST ((i,s,d,ftp_port_request pr), (InL, policy)) =
    if not_before (is_close i) (is_init i) InL ∧
      dest_port (i,s,d,ftp_port_request pr) = 21
    then ⌊((i,s,d,ftp_port_request pr)#InL,
          allow_from_to_port (net_of d) (net_of s) pr ⊕policy)⌋
    else ⌊((i,s,d,ftp_port_request pr)#InL,policy)⌋

(* FTP_CLOSE *)
|A_ST ((i,s,d,ftp_close), (InL,policy)) =
    if  ∃ p. port_open InL i p) ∧dest_port (i,s,d,ftp_close) = 21
    then ⌊((i,s,d,ftp_close)#InL,
           deny_from_to_port (net_of_of d) (net_of_of s)
           (SOME p. port_open InL i p)
           ⊕ policy)⌋
    else ⌊((i,s,d,ftp_close)#InL, policy)⌋

(* DEFAULT *)
|A_ST (p, s) = ⌊(p#(fst s),snd s)⌋
```

Please note that we require the port request message to be sent to the correct port number on the server. $SOME\ x.\ P\ x$ is the Hilbert operator usually written as ϵ and returns some x such that Px is true, and is undefined if no such x exists.

Next we model the policy. The policy that needs to applied in any given state is part of that state. Thus the policy is the application of the policy of the current state. This is formalised as follows:

```
definition applyPolicy:: (ι × (ι ↦α)) ↦α
where applyPolicy = λ(x,z). z x
```

For the combination of the policy with the two state transitions we can reuse the standard UPF operators:

```
definition FTP_TRPolicy where
FTP_TRPolicy = ((A_ST,D_ST) ⊗▽ applyPolicy) o (λ (x,(y,z)). ((x,z),(x,(y,z))))
```

Omitting the type variables, this transition policy has the following type:

packet × FWState ↦unit × FWState

which is the required kind of type of a transition policy as outlined in section 3.4.

Note that using this definition, the *observable output* of a sequence will be only the decision, as the outcome of the transition policy is only unit. When one wants to combine stateless and stateful test execution, it might be useful to have the current policy be contained in the output. This can easily be achieved by a slight adaptation of above definition:

definition FTP_TRPolicy2
where FTP_TRPolicyP2 = (λ(x,y,z). (z,(y,z))) o_f
((A_{ST},D_{ST}) $\otimes_\nabla$ applyPolicy) o (λ (x,(y,z)). ((x,z),(x,(y,z))))

Now the type of this transition policy is:

packet $\times$ FWState $\mapsto$FWPolicy $\times$ FWState

An interesting point needs to be mentioned here: both variants of transition policies are actually *second-order policies*. The policy rules specify how the policy can be transformed. This concept appears in many real-word policies and the possibility to integrate it easily into the UPF is one of the big advantages of the framework.

Modelling Voice over IP

The FTP protocol is just one simple example of how to model stateful network protocols. More complex ones are modelled using the same scheme. As an example, we present the modelling of the Voice over IP (VoIP) protocol.

VoIP is standardised by the ITU-T under the name H.323, which can be seen as an "umbrella standard" which aggregates standards for multimedia conferencing over packet-based networks.

We only consider a simplified VoIP scenario with the following seven messages, which are grouped into four subprotocols (see Figure 6.2). Note that we only consider the messages which are relevant for the caller, and not those exchanged between the callee and the gatekeeper.

- Registration and Admission (H.225, port 1719): The caller contacts its gatekeeper with a call request. The gatekeeper either rejects or confirms the request, returning the address of the callee in the latter case.
  - Admission Request (ARQ)
  - Admission Reject (ARJ)
  - Admission Confirm (ACF)
- Call Signaling (Q.931, port 1720) The caller and the callee agree on the dynamic ports over which the call will take place.

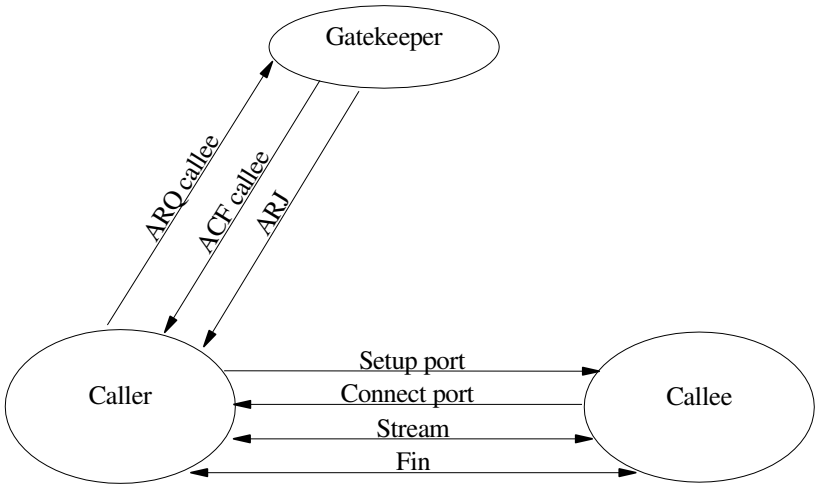

Figure 6.2: The modelled VoIP-Protocol

  - Setup port
  - Connect port

- Stream (dynamic ports). The call itself. In reality, several connections are used here.
- Fin (port 1720)

The two main differences to FTP are:

- In VoIP, we deal with three different entities: the caller, the callee, and the gatekeeper.
- We do not know in advance which entity will close the connection, and both can send stream messages.

We model the protocol as seen from a firewall at the caller; namely we are not interested in the messages from the callee to its gatekeeper. Incoming calls are not modelled either, they would require a different set of state transitions.

The content of a packet now consists of one of the seven messages and a default one. It is parametrised with the type of the address that the gatekeeper returns.

datatype α voip_msg = ARQ | ACF α | ARJ | Setup port |
Connect port | Stream | Fin | other

The state transitions for the allowance case are now more complex.

- In the case of an ARQ packet, we must ensure that we will accept the returning packet of the gatekeeper (on port 1719):

  A_{ST} ((a,c,d,ARQ), (InL, policy)) =
  $\lfloor$((a,c,d, ARQ)#InL,
  allow_from_to_port 1719 (net_of d) (net_of c) $\oplus$policy)$\rfloor$

- If the gatekeeper answers by an ARJ, we can close the channel again:

  A_{ST} ((a,c,d,ARJ), (InL, policy)) =
  if (not_before (is_fin a) (is_arq a) InL)
  then $\lfloor$((a,c,d,ARJ)#InL,
  deny_from_to_port (net_of c) (net_of d) 14 $\oplus$policy)$\rfloor$
  else $\lfloor$((a,c,d,ARJ)#InL,policy)$\rfloor$

- If the answer was positive (ACF), we allow the caller to contact the callee and to get contacted by him over port 1720:

  A_{ST} ((a,c,d,ACF callee), (InL, policy)) =
  $\lfloor$((a,c,d,ACF callee)#InL,
  allow_from_to_port (net_of_adr callee) (net_of d) 1720 $\oplus$
  allow_from_to_port (net_of d) (net_of_adr callee) 1720$\oplus$
  deny_from_to_port (net_of d) (net_of c) 1719 $\oplus$
  policy)$\rfloor$

- In the Setup message, the caller specifies the port on which he wants the connection to take place, so we need to open it for incoming VoIP messages:

  A_{ST} ((a,c,d, Setup port), (InL, policy)) =
  $\lfloor$((a,c,d,Setup port)#InL,
  allow_from_to_port (net_of d) (net_of c) port $\oplus$policy)$\rfloor$

- The same happens after the Connect message of the callee:

  A_{ST} ((a,c,d, Connect port), (InL, policy)) =
  $\lfloor$((a,c,d,Connect port)#InL,
  allow_from_to_port port (net_of d) (net_of c) $\oplus$policy)$\rfloor$

- In the FIN message, we must close all the previously opened ports. Depending on who issued the message, we need to distinguish two cases:

  A_{ST} ((a,c,d,Fin), (InL,policy)) =
  if $\exists$ p1 p2. ports_open a (p1,p2) InL then (
  if src_is_initiator a c InL
  then $\lfloor$(((a,c,d,Fin)#InL,

```
deny_from_to_port 1720 (net_of c) (net_of d) ⊕
deny_from_to_port (snd (SOME p. ports_open a p InL))
                  (net_of c) (net_of d) ⊕
deny_from_to_port (fst (SOME p. ports_open a p InL))
                   (net_of d) (net_of c) ⊕policy))⌋

          else ⌊((a,c,d,Fin)#InL,
deny_from_to_port 1720 (net_of c) (net_of d) ⊕
deny_from_to_port (fst (SOME p. ports_open a p InL))
                  (net_of c) (net_of d) ⊕
deny_from_to_port (snd (SOME p. ports_open a p InL))
                   (net_of d) (net_of c) ⊕policy)⌋)
     else ⌊((a,c,d,Fin)#InL,policy)⌋
```

- and finally the default case for all remaining network packets:

  A_{ST} (p, (InL, policy)) = ⌊(p#InL,policy)⌋

The rest is completely analogous to the FTP case: the denial state transition is straightforward, and then we apply the default UPF operators:

```
fun D_ST where
 D_ST (p,s) = ⌊s⌋

definition VoIP_Policy where
VoIP_Policy = ((A_ST,D_ST) ⊗▽ applyPolicy) o (λ (x,(y,z)). ((x,z),(x,(y,z))))
```

In modelling the VoIP protocol we can make an observation that is typical for many policy transition systems: the modelling of the state transitions is much more complex than modelling the static policies uniquely.

6.3 Unit Tests for Firewall Policies

6.3.1 General Approach to Firewall Unit Tests

In this section, we present some results from unit testing of firewall policies. The approach introduced in section 4.1 can be applied to the various firewall policy models presented in the previous section in a straightforward way. The *test specification* in its simplest form states that for every packet x, the *firewall under test (FUT)* has the same behavior as specified in the formalised policy:

```
test_spec ⋀x. FUT x = Policy x
```

Usually, this test specification is extended by predicates constraining the input space. Common predicates in the firewall scenario are:

- Wellformedness-constraints: Port numbers must be in the range 0 - 65535, IP addresses in the range 0 - 2^{32}.
- Constraints for setting dummy values: If the content or id part of a packet is not required for a test, they can be set to some dummy value.
- Constraints limiting the packets to specific origins and destinations, ensuring only packet that cross the relevant test points (e.g. the firewall under test) are generated.

These constraint predicates serve two purposes. First, by reducing the input space, they reduce the number of generated test cases as well as the time required for generating them. Second, if chosen carefully, they can improve the quality of the generated test data set significantly.

The test case generation algorithm returns a set of abstract test cases. Test data generation instantiates each of them to concrete values using constraint and random solving mechanisms. Next are two examples of test data for a filtering policy, where the first is a packet meant to be rejected by the firewall, while the second one should be passed:

1. $FUT(12, (7, 6), (172, 80), \text{content}) = \lfloor \text{deny } () \rfloor$
2. $FUT(8, (172, 12), (16, 25), \text{content}) = \lfloor \text{accept } () \rfloor$

Such test data can be exported and used as input for a tool that executes the test on a real network setup (the test driver). Overall, testing stateless firewalls is quite similar to classical unit testing of stateless software.

6.3.2 A Typical Example

At this point we present an firewall policy model. The scenario is one of a firewall between two networks that performs both filtering and Network Address Translation (NAT) tasks. As the address representation we chose the format where an address is modelled by a triple, with two integers denoting host address and port number, and an additional element denoting the protocol which is a datatype consisting of the two elements `tcp` and `udp`.

The two networks are defined as consisting of a subnetwork containing all those hosts with addresses within a given range:

definition subnet1 **where**
subnet1 = {{(a,b,c). a $>$ 1 $\wedge$a $<$ 256}}

definition subnet2 **where**
subnet2 = {{(a,b,c). a > 1001 ∧a < 1256}}

The filtering policy allows certain port numbers for each protocol and denies everything else:

definition filter :: (α,β) packet ↦unit **where**
filter =
allow_from_to_ports_prot subnet1 subnet2 tcp {21,80,443,25}⊕
allow_from_to_ports_prot subnet1 subnet2 udp {68,67,138} ⊕
allow_from_to_ports_prot subnet2 subnet1 tcp {45,346,457} ⊕
D_U

Next we define three translation rules, two performing source Port Address Translation (PAT), the other source Network Address Translation (NAT), all to a pool of addresses. All the *Adri_j* terms represent a range of integers.

definition nat1 :: (α,β) packet ↦(((α,β) packet) set) **where**
nat1 = srcPat2pool Adr1_1 Adr2_1

definition nat2 **where**
nat2 = srcPat2pool Adr1_2 Adr2_2

definition nat3 **where**
nat3 = srcNat2pool Adr1_3 Adr2_3

These rules can be combined using an override operator with a default allowance rule in the end that maps each packet to a set containing the identity element only:

definition nat **where**
nat = nat1 ⊕ nat2 ⊕ nat3 ⊕ A_f (λx. {x})

Finally, the complete policy is the parallel composition of the two sub-policies where the decision of the filtering policy is taken:

definition Policy **where**
Policy = (λ (x,y). x) o_f ((nat $\otimes_2$ filter) o (λ x. (x,x)))

Meaningful test constraints in this example could be the following three:

- accrossSubnets: Packets go from one network to the other only:

  (src x ⊏subnet1 ∧dest x ⊏subnet2) ∨
  (src x ⊏subnet2 ∧dest x ⊏subnet1)

- portPositive: All port numbers are positive values only:

$0 < \text{dest\_port x} \wedge 0 < \text{src\_port x}$

- equalProtocol: The protocol indication of the source and of the destination address must coincide:

  $\text{src\_protocol x} = \text{dest\_protocol x}$

Using this, the test specification is as follows:

test_spec
$\bigwedge$x. $[\![$ accrossSubnets x; portPositive x; equalProtocol x$]\!] \Longrightarrow$
FUT x = Policy x

and can be processed by the test case generation algorithm by issuing:

apply (gen_test_cases FUT simp: Lemmas)

where `Lemmas` contains all the required definitions of the terms contained in the policy and the test predicates.

After storing the test theorem we can apply the test data generation algorithm. As mentioned in section 2.2, this is essentially a combination of constraint solving, a random solver, and an SMT solver. All of this happens automatically. Finally, in this example, we end up with test data like the following:

FUT (6, (8, 3, udp), (1002, 21, tcp), data) =
$\lfloor$allow {(i, (s1, s2, s3), (d1, d2, d3), da).
i = 6 $\wedge$s1 = 1005 $\wedge$s3 = udp $\wedge$d1 = 1002
$\wedge$ d2 = 21 $\wedge$d3 = tcp $\wedge$da = data}$\rfloor$

6.3.3 Empirical Evaluation

In this section, we report on some empirical results about testing stateless firewall policies both with regards to the number of test data generated, as well as the required time. We will see that the examples confirm the findings in section 4.1.

The kinds of policies have been inspired by ones used in industry, but we use artificial values. There are a couple of scenarios, and in all of them we vary one or more parameter to evaluate the consequences. Unless otherwise noted, all examples use an address representation consisting of a pair of integers, the first denoting the host address, the second the port number. Furthermore, the indicated number of rules in the tables does not contain the default deny rule.

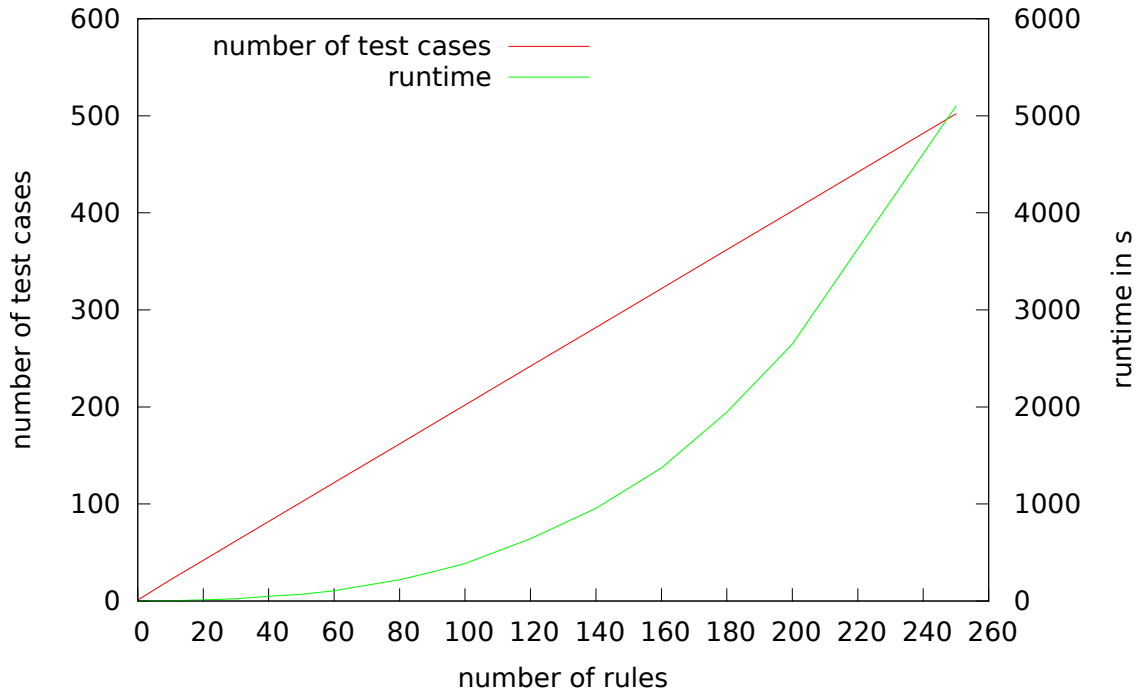

Figure 6.3: The number of test cases and the runtime increase moderately when adding similar rules.

Packet Filter with Varying Number of Rules

This scenario consists of what is often called a "personal firewall". There are two networks: a workstation and the internet. The workstation consists of one single host, with all the remaining addresses belonging to the internet. All test specifications contain a constraint limiting the tests to packets that either have their source or their destination in the workstation network, and the other somewhere in the internet. The policies are constructed by a sequence of rules allowing packets arriving on one single port on the workstation, combined using an override operator and with a default deny rule in the end.

Figure 6.3 displays the number of generated test cases in relation to the number of rules allowing certain traffic in the policy n and the required time for test case generation. Recalling section 4.1, we can expect a relatively modest increase by adding a rule, as all the rules are conceptually very similar and only differ in one part of the input (although considering two others as well). This is reflected in the empirical results: the number of generated test cases for a policy consisting of n rules is $2 * n + 2$. It is no surprise that two additional test cases are being generated for each new rule: two packets with the new port number considered, one in each of the possible two directions. The required time increases slightly faster, but remains acceptable.

Rules	10	20	30	40	50
Test Cases	22	42	62	82	102
Test Cases (Summarised)	24	44	64	84	104

Table 6.2: The number of test cases corresponds roughly to the number of rules, even if the policies are semantically very different.

As an intermediate result from this experiment, we can conclude that for policies of moderate complexity, in other words with similar rules, the coverage is as expected and also policies with a large number of rules can be processed relatively efficiently.

There is another important result to be taken from this experiment: the processing time and the number of generated test data, and consequently also the achieved coverage, does not only depend on the semantics of the model of the policy, but also of its concrete representation. For example, note that a sequence of rules allowing one port between two networks might also be represented as a single rule allowing a set of port numbers between these networks. To explore the effect of such a representation on test case generation, we adapt above experiment by always combining ten of the rules to one single rule (i.e. one rule allows a set of ten port numbers between the two networks) and run the algorithm again. The result is presented in Table 6.2. We can see that the number of test cases is similar regarding the number of rules of the policy, even though in the second case the number of rules is 10 times smaller for a semantically equivalent policy. Of course the coverage is significantly reduced here; thus it is not a general recipe to combine as many rules as possible just out of efficiency considerations.

Packet Filter with Varying Number of Networks

Next, we investigate the effect of varying another parameter: the number of networks. Again we start with a default denial policy, which is prepended by a sequence of rules using an override operator. Each of the rules allows packets to the same port, originating in a new network and destined for the Internet. Each of the networks consists of a set of addresses.

The results can be seen in Figure 6.4. We see that this time the number of generated test cases increases much more than in the previous case where the number of networks remained constant. As the time required also increases rapidly, only policies with a modest number of networks can be processed. Two of the reasons mentioned in section 4.1 contribute to the fact that enlarging the number of networks increases complexity much larger than only enlarging the number of rules. First of all, there is the fact that the number of relevant input variables is now larger. Instead of only the port number, there are two additional variables for the networks (source and destination). Furthermore,

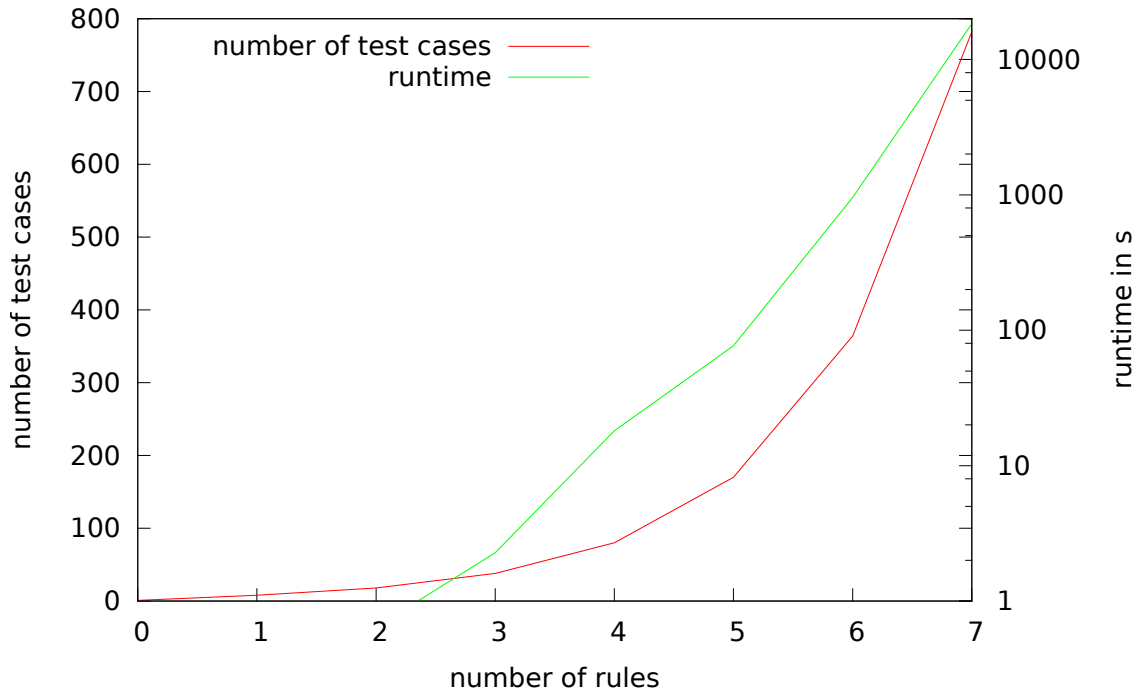

Figure 6.4: The complexity of test case generation increases quickly when adding rules of larger internal complexity

those new input variables representing sets of sets of integers are of a bigger internal complexity than the port numbers, which are modelled as simple integers only.

At this stage we already notice that the additional complexity in the rules leads test case generation to no longer scale using the naive approach. We propose a solution allowing one to deal with these kinds of policies in the next section.

We mentioned in section 4.1 that not all concepts of the policies must be unfolded before or while applying the test case generation algorithm. While this method must be applied with care in order not to reduce coverage of important aspects, this example provides one example of where it might be useful: the networks do not necessarily have to be unfolded. That way, the algorithm would only reason about membership or non-membership within a given network, and not about the concrete integer values the addresses in this network possess. Of course, the concepts must still be unfolded before test data generation, as otherwise the algorithm could not choose concrete values.

Figure 6.5 and Figure 6.6 present the outcome of generating tests for the same policies again with an increasing number of rules and networks, but where the network definitions have only been unfolded after application of the test case generation algorithm. We observe a significant drop in both the number of generated test cases, as well as in the required time. As often the concrete values of the addresses do not matter, this approach

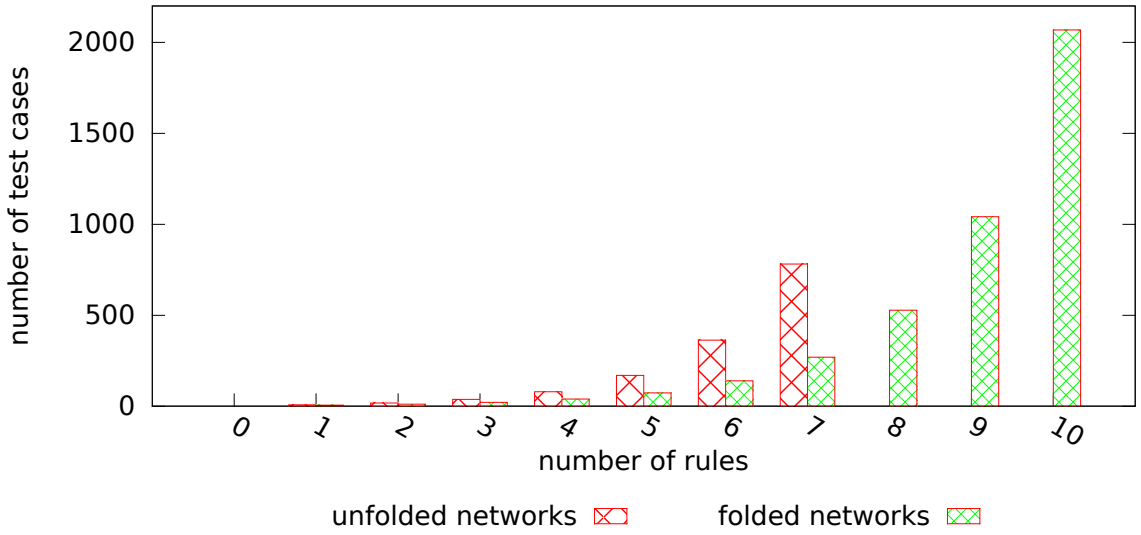

Figure 6.5: The number of test cases decreases significantly when not unfolding the networks.

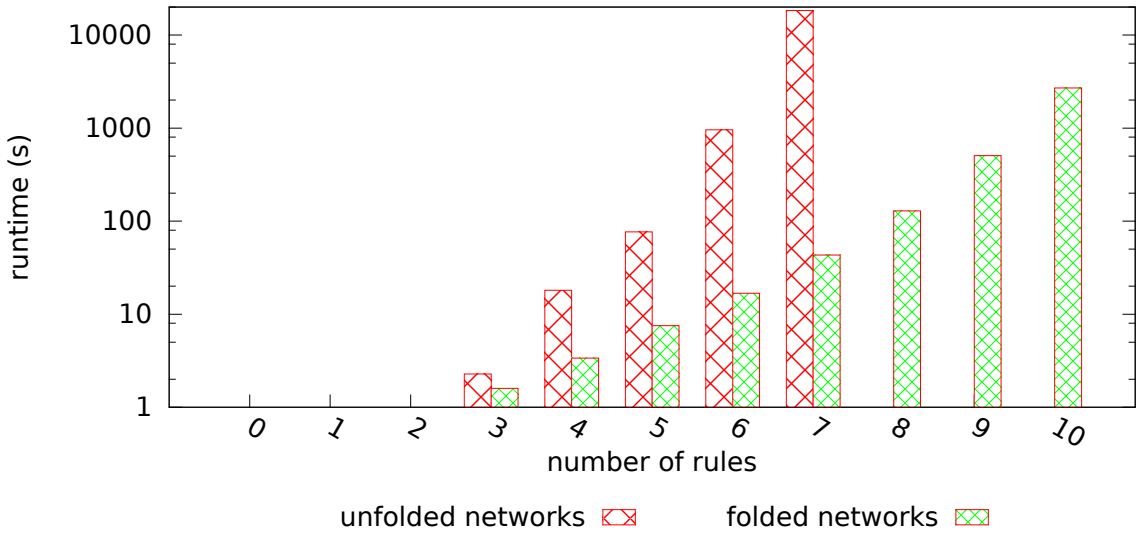

Figure 6.6: The runtime decreases significantly when not unfolding the networks.

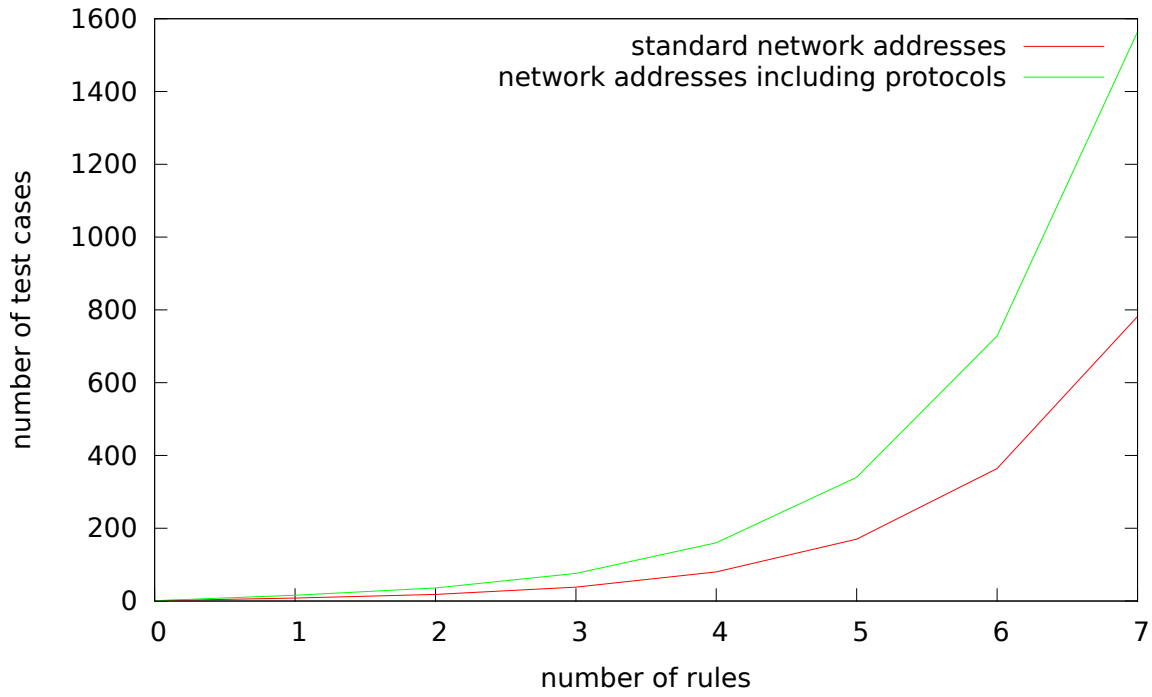

Figure 6.7: The number of test cases increases when using a more complex address representation.

is applicable in many cases. In essence, we simulate with this experiment the (partial) removal of the second effect on complexity mentioned before (internal complexity of the input variables).

Effect of Address Representation

In order to study the effect of a more complex input type representation, we carried out the first scenario of the last example with an address representation where an address is modelled by a triple, with the additional element denoting the protocol which is a datatype consisting of the two elements `tcp` and `udp`. The rules of the policy are the same as before, but additionally we restrict the packet protocol to either `tcp` or `udp` (alternating). The effect is presented in Figure 6.7 and Figure 6.8. We observe that the number of generated test cases doubles, corresponding to the two additional possibilities for each of the previous paths in the policy. Interestingly, the time complexity increases much quicker.

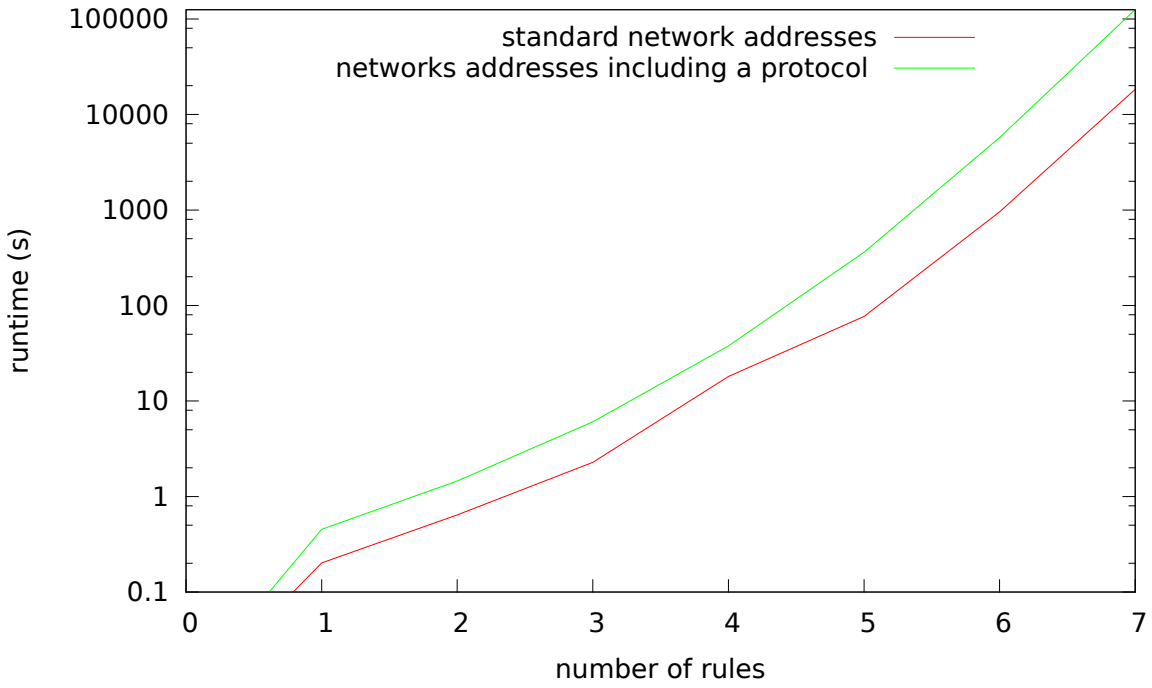

Figure 6.8: The runtime increases even more when using a more complex address representation.

Packet Filter and NAT

The next two examples evaluate the effects when testing a policy composed in parallel from two individual policies. More specifically, we explore the effect of testing a combination of a NAT and a filtering policy.

There are two subnetworks in this scenario. The filtering policy consists of a number of rules allowing packets for differing ports from the first network to the second, and denying anything else. The NAT policy consists of a number of rules, each transforming certain packets originating from the first network. All other packets remain unchanged.

The example is carried out in two variants: in the first one, the number of rules of both subpolicies increases uniformly (see Figure 6.9). In the second variant, the number of filtering rules is fixed to three and only the number of NAT rules increases (see Figure 6.10). We observe that when increasing the number of rules in both policies in a rather non-related way, the complexity increases substantially.

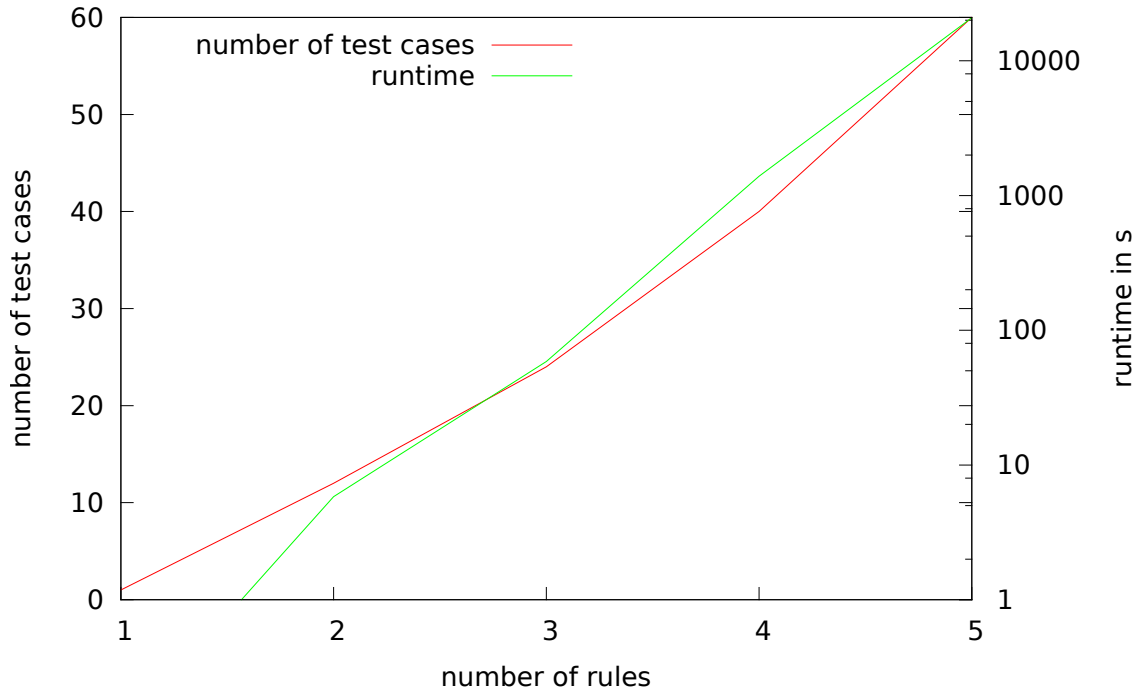

Figure 6.9: The number of test cases increases quickly when adding new rules combined in parallel

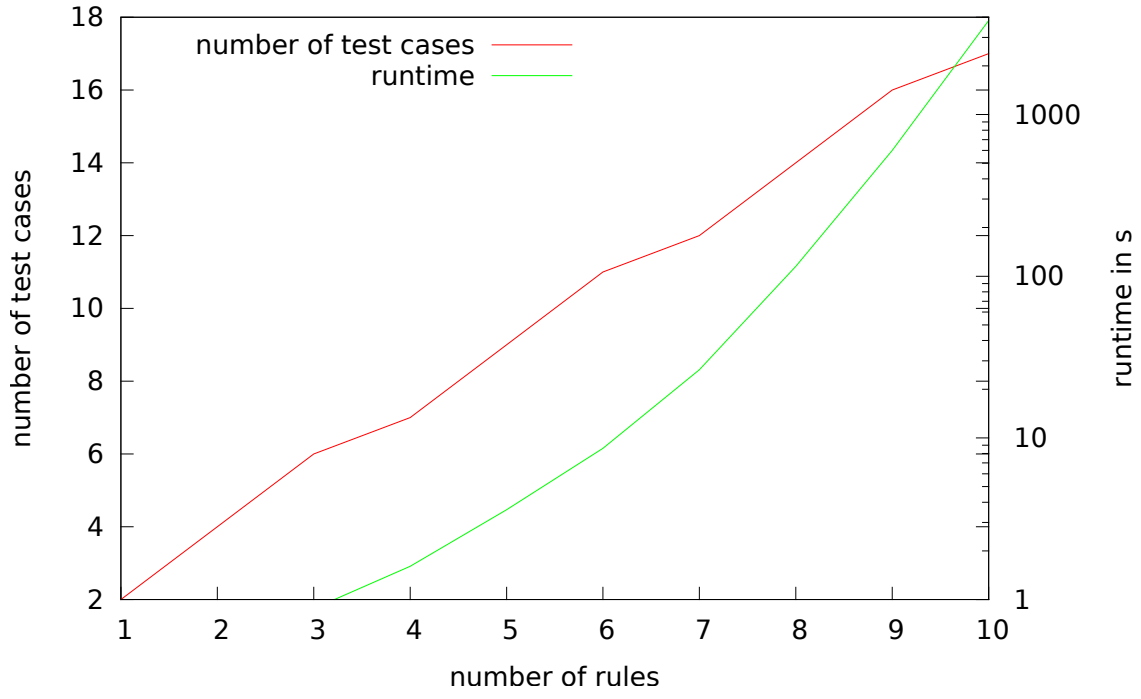

Figure 6.10: The number of test cases increases less quickly when adding new rules to only one of the policies to be combined in parallel

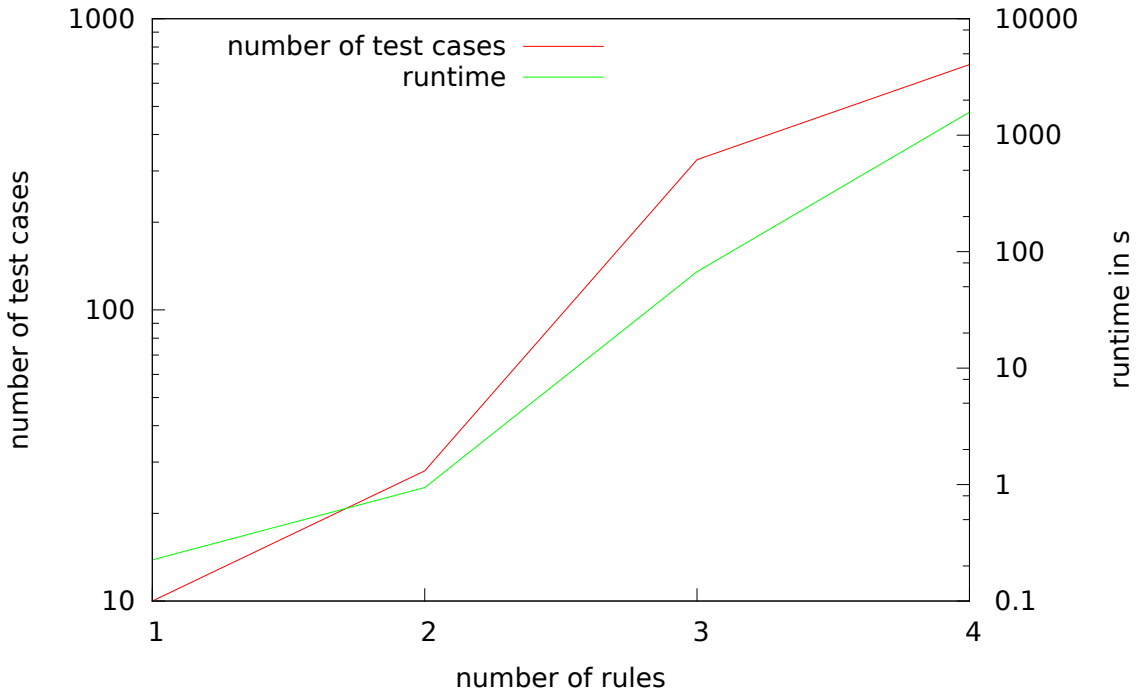

Figure 6.11: The number of test cases increases substantially when the the rules of the policy are not similar to each other.

A non-uniform Policy

So far we have considered diverse scenarios where one or more parameters were changed in a controlled manner. Here we would like to carry out an experiment with a small non-uniform policy with very diverse rules. The choice of the rules is random and thus the results of the experiment are of limited value. Furthermore, in our experience, most common real policies are rather uniform. Nonetheless, this experiment gives an indication about the effects of processing non-standard policies.

We start with the default deny rule, and then sequentially add the following rules using an override operator.

- $AllowPortTo\ subnet1\ internet\ 80$
- $DenyFromTo\ subnet2\ subnet4$
- $AllowPortRangeTo\ subnet3\ subnet1\ \{80, 180, 280\}$
- $AllowFromTo\ subnet3\ subnet4$

We do not employ any coverage- and complexity-reducing strategy. In particular we unfold all concepts for the test case generation algorithm. The results are presented in

Figure 6.11.

This small example has confirmed that by adding very diverse rules and processing them in a non-optimal manner will lead to a very deep coverage and consequently a very high complexity of the test case generation. Furthermore, please note that the relative increase in both the time required and the number of generated test cases when adding one single rule largely depends on the kind of the rule.

Summary

The empirical evaluation of the firewall policies in this section allows one to draw the following conclusions. As the firewall policies are conceptually quite similar to many kinds of policies, and as they integrate most aspects of the UPF, these results can be carried over to many other policy models.

- By default, due the fact that the test case generation algorithm uses a complete condition coverage on one hand, and because many policies are quite large and complex on the other hand, it follows that naive test case generation is only possible for small or mid-sized policies.
- If the rules the policy consists of are similar to each other, the situation improves substantially.
- Coverage and complexity can be guided by the user by using interactive theorem-proving facilities, for example by unfolding certain definitions at the right places.
- Further techniques are required to make unit testing for many real-world policies feasible. These techniques will be presented next.

6.4 Transformations of Firewall Policies

In this section, we report in more detail on one concrete, very large-scale transformation procedure for stateless firewall policies, The normalisation uses the common variant using syntactic combinators, as presented in section 4.2. For the sake of conciseness, we only allow a few core combinators.

datatype (α,β) Combinators =
  DenyAll
| DenyAllFromTo α α
| AllowPortFromTo α α β
| Conc ((α,β) Combinators) ((α,β) Combinators)

Moreover, we introduce the infix notation, _ $\oplus_S$ _, for the constructor Conc _ _. Next, we define the semantic interpretation function C from syntactic combinators to the semantic ones:

```
fun C :: (IntegerPort net, port) Combinators ⇒
         (IntegerPort,DummyContent) packet ↦unit
where
 C DenyAll = deny_all
|C (DenyAllFromTo x y) = deny_all_from_to x y
|C (AllowPortFromTo x y p) = allow_from_to_port x y p
|C  (x ⊕S y) = C x ++ C y
```

Please note that the procedures presented in the sequel also work for other semantic interpretation functions. For example, we provide examples that map the combinators to rules using other address representations as well. Additionally note that we use the override operator that applies the last applicable rule, while by default we use $\oplus$.

In the following, we first report on some elementary transformation rules, then outline in detail one notable transformation procedures, and proceed by presenting a complete normalisation procedure for stateless firewall policies consisting of a number of transformation functions applied in sequence. Finally, we show the effect of the normalisation procedure on a small example and conclude with empirical results on large-scale policies.

6.4.1 Elementary Transformation Rules

A large collection of elementary policy transformation rules can be proved correct. For example, we can show that a shadowed rule is redundant, the _ $\oplus_S$ _ operator is associative, and in many cases, commutativity holds:

$$\text{C}\ \ (\text{DenyAll} \oplus_S a) = \text{C}\ \ \text{DenyAll}$$

$$\text{C}((a \oplus_S b) \oplus_S c) = \text{C}(a \oplus_S (b \oplus_S c))$$

$$\text{C}(\text{AllowPortFromTo}\ \ x\ y\ a \oplus_S \text{AllowPortFromTo}\ \ x\ y\ b) = \text{C}(\text{AllowPortFromTo}\ \ x\ y\ b \oplus_S \text{AllowPortFromTo}\ \ x\ y\ a)$$

$$\text{C}(\text{DenyAllFromTo}\ \ x\ y \oplus_S \text{DenyAllFromTo}\ \ u\ v) = \text{C}(\text{DenyAllFromTo}\ \ u\ v \oplus_S \text{DenyAllFromTo}\ \ x\ y)$$

$$\text{dom}(\text{C}\ \ a) \cap \text{dom}(\text{C}\ \ b) = \{\} \Longrightarrow \text{C}(a \oplus_S b) = \text{C}(b \oplus_S a)$$

The proofs of these theorems are trivially to establish by the simplifier provided in Isabelle.

6.4.2 A Transformation Procedure

Here we present one particular transformation procedure. The procedure takes a list of single rules and a list defining an ordering on networks as input, and returns a list consisting of the same rules but ordered according to an ordering relation. In the next section, we present a firewall policy normalisation consisting of nine such transformations.

The sorting phase is formalised as a function mapping a list of syntactic policy operators (called combinators) to a list of combinators sorted according to the following principles:

- If there is a DenyAll, it should be at the first place.
- Rules dealing with the same set of networks should be grouped together, with the DenyAllFromTos coming after the AllowPortFromTos.

Note that this ordering is usually not unique and that its semantic correctness holds only under certain conditions.

In order to define a sorting algorithm on policy rules, we need to provide an ordering. For this, we provide a smaller relation:

```
fun smaller ::
(α, β) Combinators ⇒(α, β) Combinators ⇒(α set) list ⇒bool
where
  smaller DenyAll x l = True
| smaller x DenyAll l = False
| smaller x y l = (x = y) ∨
  (if (bothNet x) = (bothNet y)
   then (case y of (DenyAllFromTo a b) ⇒(x = DenyAllFromTo b a)
                 | _ ⇒ True)
   else (position (bothNet x) l ≤ position (bothNet y) l))
```

Here, bothNet returns the set of the source and destination network of a rule, and pos returns the position of an element within a list. The variable l in the definition above is a list of network sets. The ordering of this list determines which group of rules is treated as smaller than another. For the correctness of the transformation, the concrete ordering of this list is irrelevant, but we require that all the sets of network pairs of a policy are in this list. This requirement is formalised in a predicate called all_in_list. There are circumstances not directly related to testing, where the ordering of this list might become relevant. This would allow one to transform a policy into a semantically

equivalent one, where certain classes of rules (for example those dealing with the most frequently appearing traffic) will be considered earlier by the firewall.

We provide two versions of the sorting function. They are both slightly adapted versions of standard sorting functions from the Isabelle library, using the ordering relation above. First, we provide an insertion sort algorithm:

```
fun insort where
 insort a [] l = [a]
|insort a (x#xs) l = (if (smaller a x l) then a#x#xs else x#(insort a xs l))

fun sort where
 sort [] l = []
|sort (x#xs) l = insort x (sort xs l) l
```

Next, we have a quicksort-based sorting algorithm:

```
fun qsort where
 qsort [] l       = []
|qsort (x#xs) l = (qsort [y←xs. ¬(smaller x y l)] l) @ [x] @
                   (qsort [y←xs. smaller x y l] l)
```

The reason why we provide both options is outlined in the empirical evaluation section later. Both functions return a *sorted* list (under some assumptions, see below), where *sorted* is defined as:

```
fun sorted  where
 sorted [] l = True
|sorted [x] l = True
|sorted (x#y#zs) l = smaller x y l ∧ sorted (y#zs) l
```

All the proofs about the semantical correctness of the sorting transformation, some of which are outlined below, merely require the fact that the sorting algorithm returns a sorted list, such that we may use any sorting algorithm as long as we can prove it is actually returning a list which is sorted according to the *sorted* definition above. One of the main prerequisites for such a proof is to show that the ordering relation is transitive, reflexive, and antisymmetric. This is true as long as all the network pairs that appear in the policy are in the list l.

Next, we have to prove that the sorting transformation is semantics-preserving. We make the following assumptions about the input policy:

- singleComb: The policy is given as a list of single rules.
- allNetsDistinct: All the networks are either equal or disjoint.
- wellformed_policy1: There is exactly one DenyAll and it is at the first position.

- wellformed_policy3: The domain of an AllowPortFromTo rule is disjoint from all the rules appearing on the left of it.

In the next subsection, we show that these conditions always hold at that specific point in our normalisation algorithm. To facilitate the proofs, we further define a matching function that returns $\lfloor r \rfloor$ for the first rule r in a policy p that matches packet x and $\perp$ otherwise. To prove the semantic correctness of `sort`, that is to prove that the application of the `sort` transformation does not change the semantics of the policy, we apply an indirect approach. We show that for all packets x, and for all policies p and s that satisfy the conditions listed above, and that have the same set of rules (but not necessarily in the same order), the first rule that matches x in p is the same as in s.

theorem C_eq_Sets_mr:
assumes sets_eq: set p = set s
assumes SC: singleComb p
assumes wp1_p: wellformed_policy1 p
assumes wp1_s: wellformed_policy1 s
assumes wp3_p: wellformed_policy3 p
assumes wp3_s: wellformed_policy3 s
assumes aND: allNetsDistinct p
shows matching_rule x p = matching_rule x s

Next, we outline a proof sketch of the theorem above, which is also proven formally in Isabelle.

A case distinction over matching_rule x p results in two cases: this expression can either be $\perp$ or $\lfloor y \rfloor$:

1. $\perp$. This case is shown by contradiction. As there is a DenyAll which matches all packets, the matching rule of x cannot be $\perp$.

2. $\lfloor y \rfloor$. This is shown by case distinction on y, i.e. on the matching rule. By our definition of the Comb datatype, there are four possibilities:

   - DenyAll: If the matching rule is DenyAll, there is no other rule in p which matches, as it is necessarily the last considered rule. Thus, as the two sets are equal, there is no other rule in s that matches either. As DenyAll is in s, this has to be the matching rule in s.

   - DenyAllFromTo a b: If this is the matching rule in p, there cannot be another rule in p that matches: as the networks are disjoint and the only DenyAll is the last considered rule, the only possibility is an AllowPortFromTo a b c, for some c. However, due to wellformed_policy3, such a rule can only be on the right of the matching rule as their domains are not disjoint. But if it were on the right, the allow-rule would be the matching rule. Thus, by contradiction, there is no such rule. Since such a rule does not exist in p, it does not exist

in s either. As DenyAllFromTo a b is in s, this has to be the matching rule in s.

- AllowPortFromTo a b c: The proof of this subgoal is very similar to the previous one: The only alternative rule which could match is the corresponding DenyAllFromTo a b, but if this rule appears on the right of the matching rule, this would contradict the wellformedness conditions. Therefore, this has to be the matching rule also in s.

- $a \oplus_S b$: This case is ruled out by the wellformedness conditions.

Having proofs that any sorted list of rules conforms to the conditions if the input list does, we can establish the main correctness theorem of the sorting transformation:

```
theorem C_eq_sorted:
 assumes ail: all_in_list p l
 assumes SC: singleComb p
 assumes wp1: wellformed_policy1 p
 assumes wp3: wellformed_policy3 p
 assumes aND: allNetsDistinct p
 shows C (list2policy (sort l p)) = C (list2policy p)
```

We thus have a sorting transformation for policies that is proven to be semantics-preserving given that some well-specified conditions hold for the input policy.

6.4.3 A Normalisation Procedure

The sorting transformation is one part of a full normalisation procedure that we present next. The procedure is organised into nine phases in total.

The result of the nine phases is a list of policies, in which:

- each element of the list contains a policy that completely specifies the blocking behavior between two networks, and

- there are no shadowed rules.

This result is desirable since the test case generation for rules between networks A and B is independent of the rules that specify the behavior for traffic flowing between networks C and D. Thus, the different segments of the policy can be processed individually. The normalisation procedure does not aim at minimising the number of rules. While it does remove unnecessary ones, it also adds new ones, enabling a policy to be split into several independent parts.

We impose the following two restrictions on the input policies:

- Each policy must contain a DenyAll rule. If this restriction were to be lifted, the insertDenies phase would have to be adjusted accordingly.
- For each pair of networks n_1 and n_2, the networks are either disjoint or equal. If this restriction were to be lifted, we would need some additional phases before the start of the normalisation procedure presented below. This rule would split single rules into several rules by splitting up the networks such that they are all pairwise disjoint or equal. Such a transformation is clearly semantics-preserving and the condition would hold after these phases.

The full procedure is defined as follows:

```
definition normalise p =
 removeAllDuplicates o insertDenies o separate o
 (sort (Nets_List p)) o removeShadowRules2 o remdups o
 removeShadowRules3 o insertDeny o removeShadowRules1 o
  policy2list ) p
```

The nine transformation rules of the normalisation procedure and their preconditions are next discussed in more detail. The preconditions are necessary to establish the correctness theorem for the overall procedure:

```
theorem C_eq_normalise:
 assumes member DenyAll p
 assumes allNetsDistinct p
 shows C (list2policy (normalise p)) = C p
```

The proof of this theorem combines the local correctness results of the single transformations (e.g. the sorting phase result described in the previous section) while discharging their preconditions using invariance properties of the previous transformations. The nine transformations are:

1. **policy2list**: transforms a policy into a list of single rules. The result does not contain policies composed via $\_ \oplus_S \_$, i.e. the property singleComb is established.
2. **removeShadowRules1**: removes all the rules that appear in front of a DenyAll. The transformation preserves singleComb and establishes well-formed_policy1, stating that there is a DenyAll in front of the policy.
3. **removeShadowRules3**: removes all rules with an empty domain; they never match any packet. It maintains all previous invariants and does not establish any required new one.
4. **remdups**: removes duplicate rules. Only the last rule appearing in the list remains. As policies are evaluated from right to left, it can easily be shown that this transformation preserves the semantics.

5. **removeShadowRules2**: removes all rules allowing the traffic on a specific port between two networks, where an earlier rule already denies all the traffic between them. This function maintains the earlier invariants, and newly establishes wellformed_policy3, the invariant necessary for the sorting function to be correct. It ensures that the domain of an AllowPortFromTo rule is disjoint from all the rules appearing on the right of it.

6. **sort**: (see previous section) maintains the previous invariants and, besides the sorted invariant, also establishes an important new one: the rules are grouped according to the two networks they consider.

7. **separate**: transforms the list of single combinators into a list of policies. Each of those policies contains all the rules between two networks (in both directions). While singleComb is invalidated by this transformation, all the others are maintained. We do get two important additional properties:

   - OnlyTwoNets: Each policy treats at most two different networks.
   - separated: The domains of the policies are pairwise disjoint (except from DenyAll). This follows from the fact that the rules were already grouped.

8. **insertDenies**: is the only phase where additional rules are inserted. In front of each policy, we add the two rules denying all traffic between those two networks in both directions. As we add new rules here, the proof of semantic equivalence is relatively involved. The main part consists of proving the property that separated is maintained with this transformation. This phase is only semantics-preserving because of the initial requirement that the policy has a DenyAll. Only after this phase the traffic between two networks is characterised completely by the corresponding policy.

9. **removeAllDuplicates**: removes superfluous duplicate DenyAllFromTos introduced by the previous phase.

After the last phase, we get a list of policies that satisfies the requirements stated in the beginning of this section. Using the correctness of the transformations, it is straightforward to prove semantic equivalence of the full normalisation. In the end, the individual elements of the returned list are policies on their own and can be processed individually by HOL-TestGen. Thus, the set of test cases for the full policy is decomposed into the set of test cases of the smaller policies.

6.4.4 Example

In the following we show the effect of the normalisation procedure on a small example policy. The original policy contains 7 rules restricting the traffic between three networks:

definition Policy :: (IntegerPort net, port) Combinators **where**
Policy = DenyAll $\oplus_S$
AllowPortFromTo intranet internet 80 $\oplus_S$
AllowPortFromTo intranet dmz 443 $\oplus_S$
AllowPortFromTo dmz intranet 25 $\oplus_S$
AllowPortFromTo intranet dmz 25 $\oplus_S$
AllowPortFromTo internet dmz 80 $\oplus_S$
AllowPortFromTo internet dmz 25

After applying the normalisation procedure, we have:

1. AllowPortFromTo intranet internet 80 $\oplus_S$
   DenyAllFromTo intranet internet $\oplus_S$ DenyAllFromTo internet intranet

2. AllowPortFromTo intranet dmz 993 $\oplus_S$ AllowPortFromTo dmz intranet 25$\oplus_S$
   AllowPortFromTo intranet dmz 25$\oplus_S$
   DenyAllFromTo intranet dmz $\oplus_S$ DenyAllFromTo dmz intranet

3. AllowPortFromTo internet dmz 80 $\oplus_S$ AllowPortFromTo internet dmz 25$\oplus_S$
   DenyAllFromTo internet dmz$\oplus_S$ DenyAllFromTo dmz internet

4. DenyAll

Now, there are four policies with a total of 13 rules. The number of generated test cases shrinks from 92 to 17, the time required to generate them from 40 to about 2 seconds.

6.4.5 Empirical Results

In this section, we report on several case studies that show the benefit of policy normalisation of stateless firewall policies with respect to both the number of test cases generated and the runtime required for generating those test cases.

In a first experiment, we explored the impact of policy normalisation on the overall time required for generating test cases on policies that were randomly generated (see Table 6.3). The most important result is that while the overall number of rules can increase during normalisation, the overall time required for generating test cases is significantly smaller after normalisation, even if we include the time needed for normalisation. While for some policies, the test case generation took more than 24 hours, the required time for normalising the policy and generating test cases for the normalised policy is only a few seconds. The overall increase in the number of rules can be explained by the fact that during normalisation, segment-specific deny rules need to be inserted. The significant reduction of test case generation time can be explained by the fact that after segmentation, each segment specific policy is much smaller and less complex. As the normalisation of policies reduces the number of test cases significantly (see Figure 6.12),

		R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
Org.	Networks	2	3	4	3	4	3	4	5	5	6
	Rules	13	9	5	9	13	13	8	15	5	11
	TC Gen.(s)	4.8	38	2.4	2.8	10216	309	119	—	2.8	—
	Test Cases	84	360	78	78	3864	860	637	—	84	—
Norm.	Rules	8	14	11	10	24	15	17	28	11	25
	Segments	2	4	5	3	7	4	6	9	5	9
	TC Gen.(s)	0.5	1.4	0.5	0.8	2.3	1.6	1.4	2.5	0.6	1.9
	Test Cases	19	31	19	22	52	34	34	58	19	49

Table 6.3: Test case generation for firewall policies utilising normalisation.

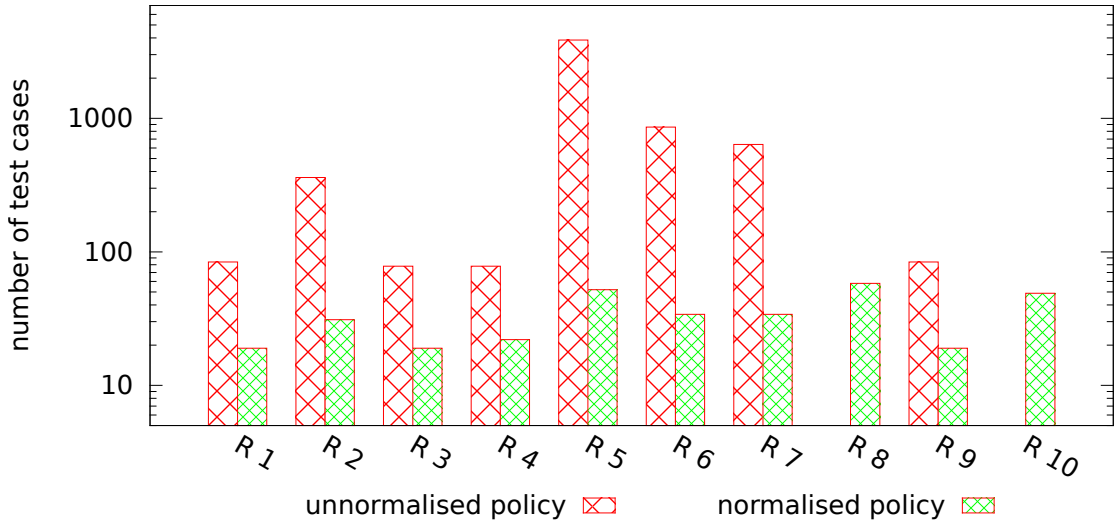

Figure 6.12: The normalisation of policies decreases the number of test cases by several orders of magnitude.

Rules	51	51	51	101	101	101	201	201	201	301	301	301
Networks	5	10	25	5	10	25	5	10	25	5	10	25
Segments	11	32	48	11	41	86	11	46	147	11	46	188
Rules	52	105	136	69	154	251	89	223	438	115	283	600
Average	5	3	3	7	4	3	9	5	3	11	6	3
Time[s] (insort)	33	83	67	102	373	590	212	1731	6460	516	2782	34248
Time[s] (qsort)	31	112	522	91	318	1593	206	765	7960	453	1198	17355
Test Cases	124	220	265	175	340	496	235	532	877	313	713	1237

Table 6.4: The size of a normalised policy mainly depends on the number of rules in relation to the number of networks.

the time needed for executing a test on an implementation is also reduced substantially.

The generation of test cases for larger policies in reasonable time is only possible after normalisation. Thus, we investigated the effect of normalisation in isolation. Table 6.4 shows the normalisation results for randomly generated policies with different sizes both in the number of rules and the number of networks covered (see also Figure 6.13 and Figure 6.14). All policies start with a DenyAll rule and one in every five rules is a DenyAllFromTo. If the number of rules is small related to the number of networks, the number of rules after normalisation increases quite significantly as there are a lot of additional DenyAllFromTo's to be inserted. In the opposite case, it decreases a lot, as many of the AllowPortFromTo rules will be removed during normalisation. The time needed for the normalisation primarily depends upon the number of networks and on the number of rules.

This experiment has been carried out with both sorting procedures. We observe that while insertion sort is faster on the smaller scenarios, quick sort is faster with the bigger ones. It must be noted that the sorting phase is by far the most time-consuming one.

The table also presents the number of test cases generated by applying the test case generation algorithm to the normalised policies. The time required remains very low compared to the time used for normalisation. Even for the biggest policy, this requires less than three minutes for both test case and test data generation combined.

As a last measurement, to present the effect of normalising parallely composed policies, we have defined a couple of policies defined as parallel composition of a filtering and a NAT policy. The normalisation is transferred over the composition operator, as outlined in section 4.2. The filtering rules are defined to allow each a distinct port between two

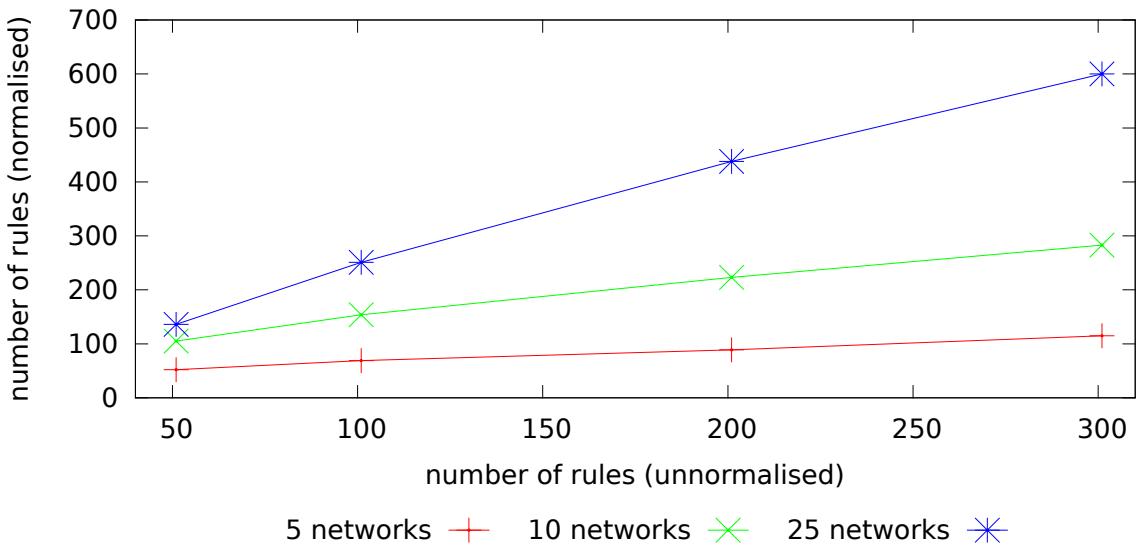

Figure 6.13: The size (number of rules) of a policy after normalisation increases with both the number of rules in the normalised policy and the number of networks.

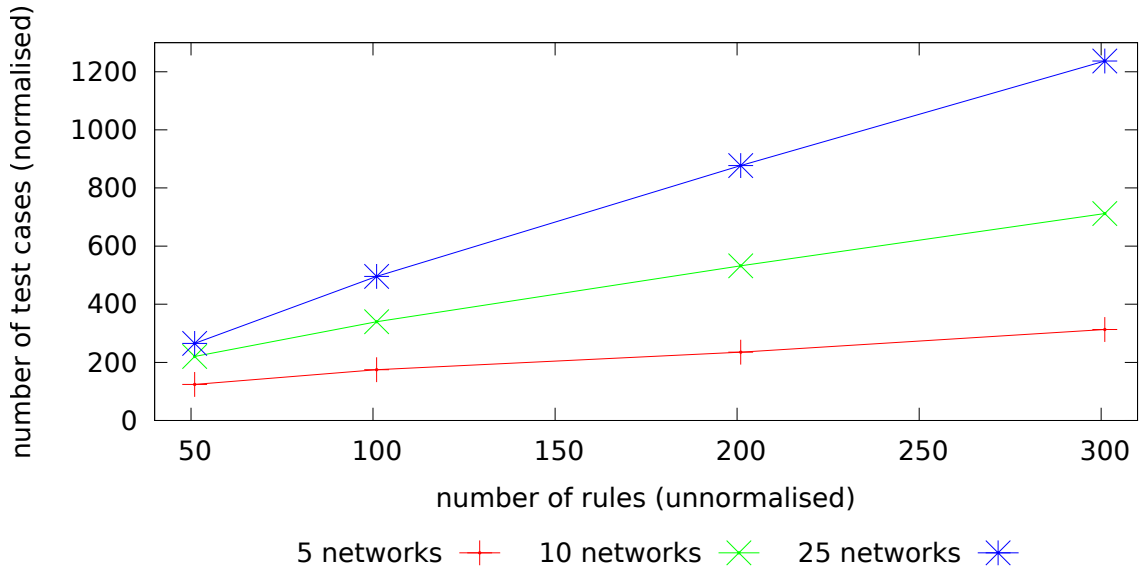

Figure 6.14: The number of test cases of a policy after normalisation increases with both the number of rules in the normalised policy and the number of networks.

distinct networks. For each filtering rule, there is a corresponding NAT rule performing source address translation to a pool of addresses, rewriting all packets having the source in the same network as the source of the corresponding filtering rule. Each policy of the experiment is constructed by an override composition of both policies individually (including a default rule for each), and performing the parallel composition in the end only. Recalling the previous section, naive test case generation for such a policy would not scale. For this reason, we apply the normalisation procedure for filtering policies presented before, which distributes over to the combined policy. The results of applying the normalisation on these policies are presented in Figure 6.15, Figure 6.16, and Table 6.5. The number of rules is the number for each subpolicy. We see that we make large gains in efficiency, which could even be increased if we would also apply a transformation procedure on the NAT part of the policy.

As a summary of the empirical evaluations of this and the previous section, we can make the following observations:

- Testing firewall policies scales well for policies with rather similar and not very complex rules,
- but does not realistically work for large-scale policies with complex and not similar rules.

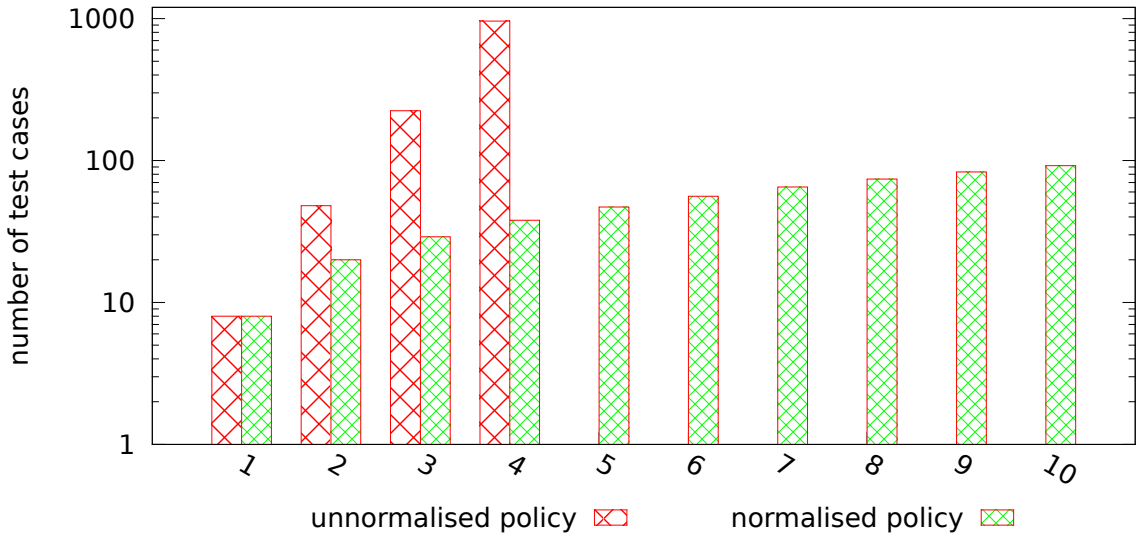

Figure 6.15: The number of test cases decreases significantly for transformed NAT/filtering policies (logarithmic scale)

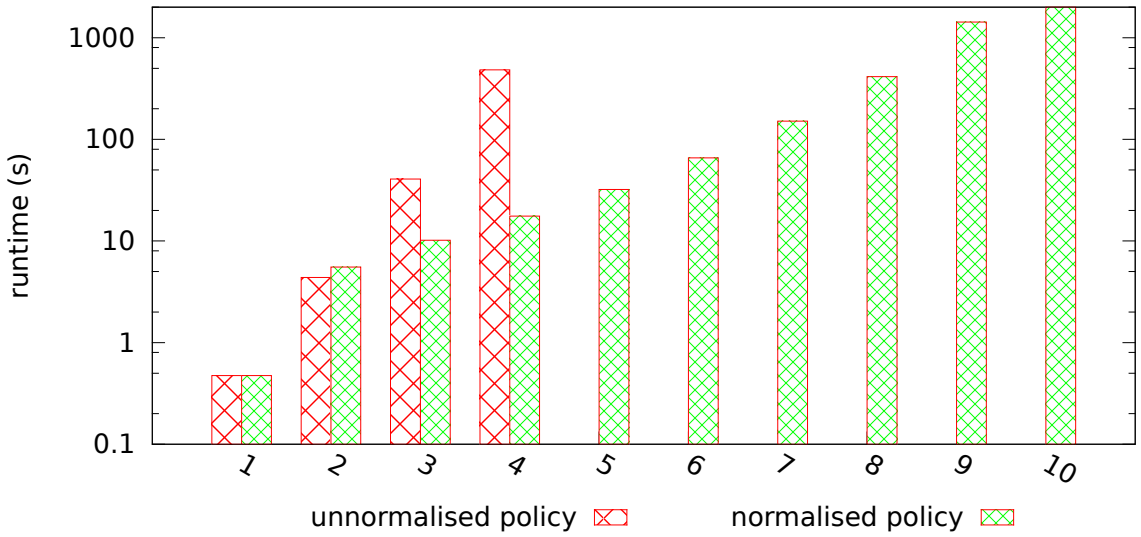

Figure 6.16: The runtime decreases significantly for transformed NAT/filtering policies (logarithmic scale)

Rules	1	2	3	4	5	6	7	8	9	10
Test Cases (normalised)	8	20	29	38	47	56	65	74	83	92
Test Cases (original)	8	48	224	960	-	-	-	-	-	-
Runtime (normalised)	1	5	10	18	32	66	151	414	1427	6225
Runtime (original)	1	4	41	483	-	-	-	-	-	-

Table 6.5: The number of test cases and the runtime of parallely composed policies: application of transformation procedures increases efficiency substantially.

- Firewall policy normalisation allows processing of policies of the latter kind by transforming them into a set of non-overlapping policies of the first kind.
- As the time required for normalisation is negligible compared to the time required for generating test cases for non-normalised policies, normalisation can be enabled by default when testing firewall policies.

All in all, the application of the policy transformation technique to the domain of firewall policies allows for a relatively efficient testing technology for a wide range of large-scale policies as encountered in practice. Of course, the coverage regarding the original specification is reduced, and consequently also the probability of detecting failures. However, we believe that only those parts from the specification have been removed from being tested that are very unlikely to contain any failures, by splitting the policy into independent parts. Our small experiments also provided us with evidence that in practice the ability to detect failures is not reduced significantly, and overall effectiveness of the testing procedure has been improved substantially.

6.4.6 Boundary Value Testing for Firewall Policies

In this section, we briefly report on a technique allowing one to add support for boundary value testing for firewall policies using policy transformation techniques.

As an example, suppose we want to add support for testing the boundaries as defined by the port numbers of the policies. For example, if there is a rule specifying port 14, we additionally want to have test cases about ports 13 and 15. Using the general policy transformation approach, this requires us to write a function adding new rules to the policy and prove its correctness.

To support boundary cases for networks, a similar approach can be employed: for each network used in the policy, define new networks which are the desired boundary cases, and add any rule about these networks after the default rule.

There are two options how a transformation procedure adding support for this could be implemented: either such additional rules are added for each port appearing in the original policy, or the user can provide a list of port numbers and network pairs to be added. In the first case, it is best to use the syntactic combinators also used in the previously presented normalisation procedure for stateless firewalls.

As an example of a function inserting new rules enabling tests to be generated for boundary cases, consider the following function inserting a number of new rules, where the user supplies a list of tuples of a source network, a destination network and a port number. This list could be generated automatically, to make the whole procedure automatic.

```
fun addBoundaryRule where
 addBoundaryRule (x#xs) P =
    addBoundaryRule xs (P ⊕
   (allow_from_to_port (snd (snd x)) (fst x) (fst  (snd x))))
|addBoundaryRule [] P = P
```

As always we need to prove semantical correctness of this function. This can be proved under the assumption that the original policy is total or that the domain of the inserted rules is a subset of the domain of the original policy.

```
lemma
⟦dom P = UNIV; FUT x = addBoundaryRule y P x⟧ ⟹FUT x = P x
```

This lemma can be applied in a test specification (before applying test case generation) by the following command:

```
 apply (rule_tac y = [(subnet1, subnet2, (14::port))]  in boundaryFUT)
```

6.5 Sequence Tests for Stateful Firewall Policies

In this section, we present several sequence testing scenarios for firewall policies. We first present the testing of FTP protocols, followed by the testing of the more complex VoIP protocol and finish with testing possible interleavings of the two protocols.

6.5.1 Testing FTP

In section 6.2, we presented a model of the FTP protocol. Here we show how this model can be used for the generation of test cases.

First we specify the test purpose, i.e. define the test specification automaton denoting acceptable input traces. As usual, producing sequences of more or less random inputs

(i.e. packets) is usually not what one wants to test against. There are a couple of standard test purposes when testing stateful network protocols:

- Tests for one correct protocol run.
- Tests for several correct protocol runs sequentially.
- Tests for several correct protocol runs interleaved.
- Tests for one or several illegal protocol runs.
- Test for protocol runs interspersed with packets that do not belong to a protocol run.

In the following, we present how to model correct protocol runs. Such a run can be in any of four different states, which is modelled as a datatype:

datatype ftp_states = S0 | S1 | S2 | S3

The automaton denoting all traces representing one correct protocol run can be modelled using a recursive function definition as follows. Note that as input it takes to addresses (of the client and the server), an id (representing the unique id of all packets of the trace), and a port number (the one the client chooses the packets to be sent to).

fun is_ftp :: ftp_states ⇒ IntPort ⇒ IntPort ⇒
id ⇒ port ⇒ (IntPort,ftp_msg) history ⇒ bool
where
is_ftp H c s i p [] = (H=S3)
|is_ftp H c s i p (x#InL) = (snd s = 21) ∧λ(id,sr,de,co). id = i ∧ (
(H=S0 ∧sr = c ∧de = s ∧co = ftp_init ∧ is_ftp S1 c s i p InL) ∨
(H=S1 ∧sr = c ∧de = s ∧co = ftp_port_request p ∧is_ftp S2 c s i p InL) ∨
(H=S2 ∧sr = s ∧de = (fst c,p) ∧ co= ftp_data ∧is_ftp S2 c s i p InL) ∨
(H=S2 ∧sr = c ∧de = s ∧co = ftp_close ∧ is_ftp S3 c s i p InL)) x

The set of legal traces according to this automaton is next defined as:

definition
NB_ftp s d i p = {x. (is_ftp S0 s d i p x)}

Next, we define the setting of the tests. In this case this is essentially the starting state. The firewall under test is between two networks, one called intranet the other Internet. We assume the initial policy to only allow FTP sessions (started using port 21, the control port of the FTP protocol) from the intranet to the Internet:

definition ftp_policy :: (IntPort,ftp_msg) FWPolicy **where**
ftp_policy = allow_from_to_port intranet internet 21 $\oplus D_U$

Depth	3	4	5	6	7	8	9	10
Test Cases	1	2	3	4	5	6	7	8
Runtime (s)	1	4	12	52	225	988	4328	18898

Table 6.6: Number of test cases and runtime for correct runs of the FTP protocol

Using this, the starting state for the test sequences is defined as the empty trace and the just defined policy:

definition
σ_0 = ([],ftp_policy)

As the final preparation step, we define a predicate on input traces specifying that they need to be in the set NB_ftp, and furthermore that the used host addresses are in the specified two networks.

definition accept_ftp t =
∃ c s i p. t ∈ NB_ftp c s i p ∧ is_in_intranet c ∧ is_in_internet s

where the predicates (isInIntranet and isInInternet) are used for checking if an address is within the specific subnet.

Now we are ready to specify the test specification. `FTP_TRPolicy` is the state-exception monad of the FTP transition policy introduced in section 6.2:

test_spec
⟦length t = length X; accept_ftp t⟧ ⟹
σ_0 ⊨(os ← mbind t FTP_TRPolicy; return (os=X)) ⟶
FUT σ_0 t = X

The number of generated test data of this specification depends on the maximum length to be explored. In order to get at least one valid test data, the depth must be at least three. Increasing it by one will also generate one more test data, as there is only one possibility conforming to the test specification. In Table 6.6, we also present the runtime for generation for each test case generation depth.

An example test data generated, here of length five, is as follows:

```
FUT ([], ftp_policy)
  [(10, (3, 0), (7, 21), ftp_init ),
   (10, (3, 0), (7, 21), ftp_port_request 2),
   (10, (7, 21), (3, 2), ftp_data), (10, (7, 21), (3, 2), ftp_data),
   (10, (3, 0), (7, 21), ftp_close)] =
[allow (), allow (), allow (), allow (), allow ()]
```

Other test specification automata can be modelled similarly. As one example, we present how to model an interleaving of two correct FTP protocol runs. The packets are mapped to a protocol run by their identifier. Using the same definition for the automaton as before, we may define the acceptable input traces by the following two definitions. While the first one defines the set of all legal traces given two identifiers, client, and server addresses, and ports to be requested, the second checks if a given trace is in that set with additional constraints on those variables. The definition `packet_with_id x i` filters the trace x by keeping only those packets having identifier i.

definition
ftp_2_interleaved ::
id ⇒ id ⇒ IntPort src ⇒ IntPort dest ⇒ port ⇒
IntPort src ⇒ IntPort dest ⇒ port ⇒
(IntPort,ftp_msg) history set **where**
ftp_2_interleaved i1 i2 s1 d1 p1 s2 d2 p2 =
{x. (is_ftp S0 s1 d1 1 p1 (packet_with_id x i1)) ∧
(is_ftp S0 s2 d2 2 p2 (packet_with_id x i2))}

definition accept2_ftp :: id ⇒ id ⇒ (IntPort, ftp_msg) history ⇒ bool **where**
accept2_ftp i1 i2 t =
∃ c1 s1 p1 c2 s2 p2. t ∈ ftp_2_interleaved i1 i2 c1 s1 p1 c2 s2 p2 ∧
is_in_intranet c1 ∧ is_in_internet s1 ∧ (snd s1) = 21 ∧
is_in_intranet c2 ∧ is_in_internet s2 ∧ (snd s2) = 21 ∧
p1 ≠21 ∧ p2 ≠21 ∧ i1 ≠ i2

Test case generation works as before, however the depth must be increased to at least six (eight if we want two full protocol runs each including at least one data packet). The number of generated test cases is now much larger, as all possible interleavings are created. The results are presented in Table 6.7. Essentially, the following formula calculates the number of possible interleavings of maximum length n of two protocols each of length at least m:

$$f(n,m) = \sum_{i=2*m}^{n} \left(2^i - 2 \sum_{i=1}^{m-1} \binom{n}{i} \right) \tag{6.1}$$

If $n < 2 * m$, there is no valid input trace and consequently no test cases are being generated.

For the case of $m = 3$, this boils down to:

$$f(n) = \sum_{i=6}^{n} (2^n - n^2 - n - 2) \tag{6.2}$$

which is 20 for length 6, 90 for length 7, and 272 for length 8. This corresponds to the number of generated test cases for the interleaved FTP protocol as shown in Table 6.7.

Depth	6	7	8	9
Test Cases	20	90	272	692
Runtime (s)	105	488	2483	12926

Table 6.7: Number of test cases and runtime for two interleaved correct runs of the FTP protocol

6.5.2 Testing VoIP

Testing the Voice over IP (VoIP) protocol as modelled in section 6.2 works very similar as testing the FTP protocol. As before, we model the networks and the initial state. This time however, we have to model three entities: in addition to the intranet and the Internet there is also a gatekeeper. Due to the more complex nature of the VoIP protocol, the corresponding test specification automaton is also more complex. The main complication arises from the fact that both the Stream as well as the Fin message may originate from either side, and furthermore the connection can also be closed prematurely by an ARJ message.

```
fun is_voip :: voip_states ⇒ address ⇒ address ⇒ address ⇒ id ⇒ port ⇒
                port ⇒ (IntPort, address voip_msg) history ⇒bool
where
 is_voip H s d g i p1 p2 [] = (H = S5)
|is_voip H s d g i p1 p2 (x#InL) =
  λ (id,sr,de,co). id = i ∧
(H = S0 ∧sr = (s,1719) ∧ de = (g,1719) ∧ co = ARQ ∧
    is_voip S1 s d g i p1 p2 InL) ∨
(H = S1 ∧sr = (g,1719) ∧ de = (s,1719) ∧ co = ARJ ∧
    is_voip S5 s d g i p1 p2 InL) ∨
(H = S1 ∧sr = (g,1719) ∧ de = (s,1719) ∧ co = ACF d ∧
    is_voip S2 s d g i p1 p2 InL) ∨
(H = S2 ∧sr = (s,1720) ∧ de = (d,1720) ∧co = Setup p1 ∧
    is_voip S3 s d g i p1 p2 InL) ∨
(H = S3 ∧sr = (d,1720) ∧de = (s,1720) ∧ co = Connect p2 ∧
    is_voip S4 s d g i p1 p2 InL) ∨
(H = S4 ∧sr = (s,p1) ∧ de = (d,p2) ∧co = Stream ∧
    is_voip S4 s d g i p1 p2 InL) ∨
(H = S4 ∧sr = (d,p2) ∧de = (s,p1) ∧ co = Stream ∧
    is_voip S4 s d g i p1 p2 InL) ∨
(H = S4 ∧sr = (d,1720) ∧de = (s,1720) ∧ co = Fin ∧
    is_voip S5 s d g i p1 p2 InL) ∨
(H = S4 ∧sr = (s,1720) ∧ de = (d,1720) ∧co = Fin ∧
    is_voip S5 s d g i p1 p2 InL) x
```

Depth	2	3	4	5	6	7
Test Cases	1	1	1	3	7	15
Runtime (s)	1	5	44	458	4740	48136

Table 6.8: Number of test cases and runtime for correct runs of the VoIP protocol

The definitions for the set of traces corresponding to that automaton and the one for the acceptance predicate on the traces are as in the FTP testing scenario. Finally, the test specification is as follows:

test_spec $[\![$length t = length X; accept_voip t$]\!] \Longrightarrow$
$\sigma_0 \models$(os $\leftarrow$mbind t VOIP_TRPolicy; return (os=X)) $\longrightarrow$
FUT σ_0 t = X

In contrast to the FTP case, where according to this test specification there is only one valid protocol run for a given length, the VoIP protocol will generate several protocol runs. This corresponds to the fact that in several steps of the automaton several actions are possible.

The minimal protocol length is two, corresponding to the case when the gatekeeper rejects the connection request. To go into the other branch, at least five packets need to be generated, and to have at least one `Stream` packet, the minimum length is six. In Table 6.8 we present the number of generated test sequences and the runtime according to the test case generation depth. Recalling section 4.3, we see that the experiments confirm the formulae presented there. First, we transform the test specification automaton into two distinct ones: the first one is applied whenever the request is rejected and the second one when it is confirmed.

The number of test cases for the first sub-automaton, $T2_1$, can be calculated by $k_1 + k_2 = 1 * 1 = 1$.

For the second sub-automaton, we can apply Equation 4.7. For any length smaller than 5, no traces can be generated. Next we have that $T2_5 = 1 * 1 * 1 * 1 * 2 = 2$, and for all $i > 0$, $T2_{5+i} = 1 * 1 * 1 * 1 * 2^i * 2 = 2^{i+1}$.

Summing up, the number of test traces for the VoIP protocol given depth d where $d > 4$ can be calculated by:

$$1 + \sum_{i=5}^{d} 2^{i-4} \tag{6.3}$$

and 1 when $1 < d < 5$, which corresponds to the numbers in the table.

6.5.3 Testing Interleavings of FTP and VoIP

Next we present how parallel executions of different protocols can be tested. In more detail, we want to test arbitrary interleavings of FTP and VoIP protocol executions. To do this, we define a new datatype denoting the possible message contents, integrating the possibilities from both the FTP and the VoIP protocols:

```
datatype ftpvoip =
ARQ | ACF int | ARJ | Setup port | Connect port | Stream | Fin |
ftp_init | ftp_port_request port | ftp_data | ftp_close |
other
```

The state transitions are defined individually for each protocol; in fact we can reuse the existing ones. They can be combined in parallel, unifying the inputs and outputs:

```
definition ST_A ::
  ((IntPort, ftpvoip) history, IntPort, ftpvoip) FWStateTransition
where
 ST_A = (λ(x,x). ⌊x⌋) o_m
  (FTP_ST_A ⊗_S VOIP_ST_A o (λ (p,x). (p,x,x)))

definition FTP_ST_D ::
  ((IntPort, ftpvoip) history, IntPort, ftpvoip) FWStateTransition
where
 ST_D = (λ(x,x). ⌊x⌋) o_m
  (FTP_ST_D ⊗_S VOIP_ST_D o (λ (p,x). (p,x,x)))
```

The same way as before, the state transitions are combined with the filtering policy to a transition policy monad:

```
definition FTPVOIP where
 FTPVOIP = p2M (
   ((ST_A,ST_D) ⊗_▽ applyPolicy) o (λ (x,(y,z)). ((x,z),(x,(y,z)))))
```

We reuse the two test specification automata used for testing the individual protocols, and combine them into an automaton allowing any possible interleavings of the two automata:

```
definition ftp_voip_interleaved ::
  IntPort src ⇒ IntPort dest ⇒ id ⇒ port ⇒
  address ⇒ address ⇒ address ⇒ id ⇒ port ⇒ port ⇒
  (IntPort, ftpvoip) history set
where
ftp_voip_interleaved s1 d1 i1 p1 vs vd vg i2 vp1 vp2 =
      {x. (is_ftp FS0 s1 d1 i1 p1 (packet_with_id x i1)) ∧
          (is_voip V0 vs vd vg i2 vp1 vp2 (packet_with_id x i2))}
```

Depth	5	6	7	8
Test Cases	10	25	46	186
Runtime (s)	139	1214	11867	119908

Table 6.9: Number of test cases and runtime for interleaved runs of the FTP and of the VoIP protocol

Assuming the same networks as before, the test specification automaton can be refined to specific values:

definition accept_ftpvoip **where**
accept_ftpvoip i1 i2 t =
∃ s1 d1 p1 vs vd vg vp1 vp2.
t ∈ ftp_voip_interleaved s1 d1 i1 p1 vs vd vg i2 vp1 vp2
∧ is_in_intranet vs ∧ is_in_internet vd ∧ is_gatekeeper vg
∧ is_in_intranet (fst s1) ∧ is_in_internet (fst d1) ∧ (snd d1) = 21
∧ i1 ≠ i2

Finally the test specification is as follows, where $ids1, 2$ restricts the ids of the packet to be either 1 or 2.:

test_spec ⟦length t = length X; accept_ftpvoip 1 2 t; ids {1,2} t⟧ ⟹
σ_0 ⊨(os ←mbind t FTPVOIP_TRPolicy; return (os=X)) ⟶
FUT σ_0 t = X

In Table 6.9 we present both the number of generated test cases for a given depth, as well as the required time for generating them.

6.6 Test Execution for Firewall Policies

For testing firewall policies, we have developed a domain-specific test execution tool, which we briefly present in the following.

An abstract architecture for the Firewall Testing System is shown in Figure 6.17. The interface with the Firewall Under Test (FUT) is via a number of "Probes" that are able to inject and/or intercept packets into/on the local area networks (LANs) to which they are attached. The firewall and probes are shown as being attached to three such LANs, though this number is arbitrary. The four main modules correspond to the essential testing stages, of which the first two have been presented before.

1. Test Specification

2. Test Data Generation
3. Test Execution
4. Test Analysis

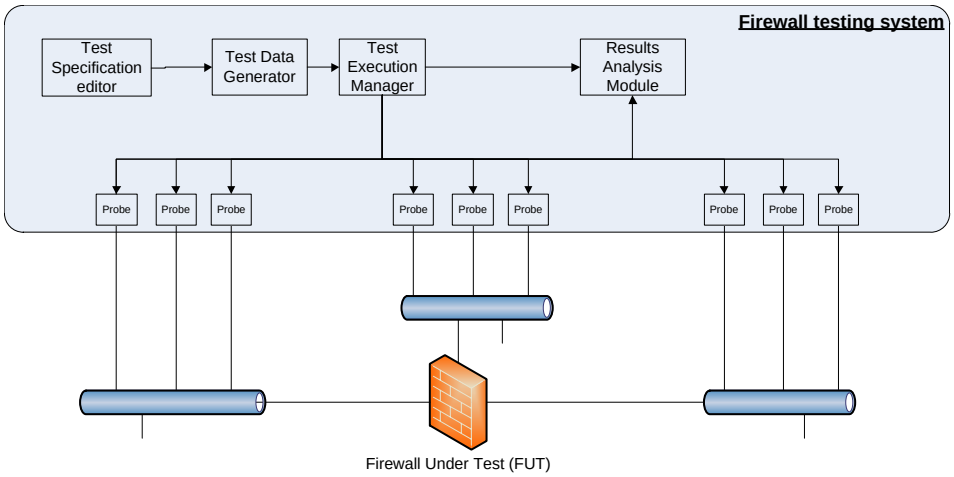

Figure 6.17: Architecture of the Firewall Testing System

The Test Execution Manager takes the generated test data as input and rewrites it as instructions that are then distributed to the Probes for execution. A description of expected results is also generated for use by the Results Analysis Module. This rewriting steps also performs the necessary input and output mappings. Here this mainly involves the mapping of the addresses and the networks of the model to the real IP addresses and networks.

Probes have two basic functions: in the "injector" role they send out packets according to the instructions, and in the "sniffer" role they report packets they observe.

One of the main architectural units of the prototype is the endpoint. An endpoint is a combination of a network interface and a set of probes using it to sniff and/or insert packets. A single machine (virtual or real one) with multiple interfaces can host several endpoints. However an injector and a sniffer on the same interface count as one endpoint only — in most cases, each endpoint contains both. The endpoints should be connected as close to the firewall under test as possible and emulate the whole network which is in normal operation connected to the respective interface of the firewall under test. In a concrete implementation, the endpoints can either all be on the same virtual machine, or on dedicated machines each.

The Probes log events corresponding to the sending and sniffing of packets. This information is made available for evaluation by the Results Analysis Module. The Results Analysis Module takes as input files containing descriptions of the expected results (produced by the Test Data Generator), the event logs recording the packets actually having

been sent and received, and calculates and prints the test results.

We have applied this framework to a number of different kinds of firewalls. Regarding efficiency, it can be noted that executing a test set consisting of hundreds of test data is normally done in less than a minute. This number depends on the concrete characteristics of the network under test, but is expected to be negligible in most cases.

6.7 Summary

In this chapter, we have presented the instantiation of the Unified Policy Framework (UPF) with the domain of firewalls and networks and its application to conformance testing. We have presented a model of stateless packet filters, network address translation rules, as well as of stateful firewalls controlling execution of protocols like FTP and VoIP. We have provided an empirical evaluation of a range of different firewall policies.

We have outlined that our test case generation approach scales well with policies consisting of rather simple and uniform rules, but does not scale when having complex and non-uniform rules. We have presented a solution to this problem by providing a normalisation technique for firewall policies instantiating our policy transformation methodology. This normalisation transforms a complex policy into a set of non-overlapping policies of a simpler kind, thereby providing a well-scaling test generation technology for most kinds of firewall policies as encountered in practice.

The generated test data are used as input to a domain-specific test execution framework for testing firewall policies. The architecture of such an execution framework consists of a number of Probes that are able to inject and intercept packets into the local area networks (LANs) to which they and the firewall under test are attached. We have applied this framework to a number of different kinds of firewalls and the initial experiments displayed a very good ability of detecting faults in the firewall configuration.

7 Related Work

In this chapter, we present related work relevant for this thesis. We are not aware of any framework integrating all the aspects we provide: a uniform modelling framework for a wide range of security policies, the techniques for model-based unit and sequence testing from these models, verified policy transformation procedures, and the means to automatically execute these tests. However, a considerable amount of literature and tools exist for most of the individual parts, which we present in the following. We start by presenting some other policy modelling frameworks, followed by generic model-based testing tools, with a special focus on those allowing for testing for policy conformance. Next, we present work related to our concept of policy transformation procedures. Finally, we conclude by describing relevant related work concerning the two instantiations of our framework presented in this thesis.

7.1 Related Policy Modelling Frameworks

In this section we compare the Unified Policy Framework with some other modelling languages.

Pbel

Pbel [BH11] is "an expressive access control policy language based on the operators of Belnap logic". We share many design principles, mainly:

- Important focus on generic combination operators, where a large policy is modelled as a composition of its sub-policies
- Core language that should be independent from concrete application domains
- The possibility to not only express granting of of rights, but also prohibitions (explicit denials)

On the other hand, the main purpose of the two frameworks is distinct: PBel is focused on policy analysis, the UPF on test case generation.

PBel is based on Belnap's four-valued logic [Bel77]. There are four possible decision values: undefined ($\bot$), grant ($\mathbf{t}$), deny ($\mathbf{f}$), and conflict ($\top$), which are structured in a bilattice. A number of generic operators are then defined over this structure. The domain-specific parts are abstracted away using request predicates.

There are two main differences between PBel and the UPF:

- In PBel, conflicts can be made explicit.
- The UPF is tightly integrated with modelling system behaviour and provides a possibility to model stateful policies directly. For this reason, in the UPF the output of a policy is a decision and an outcome, while PBel only provides a (richer) decision value

From these two differences it follows that none of the two frameworks is strictly more powerful than the other: both allow specification of policies that cannot be modelled in the other framework. However, a large part of PBel can be easily transformed into UPF policies. PBel introduces the notion of a conflict-free policy, and a subset of the language which only leads to conflict-free policies. This sublanguage does not contain the $\top$ value and the decisions are thus comparable to the UPF ones. Many operators which are part of this sublanguage can be mapped to the UPF. However, the standard operators are orthogonal to the the UPF ones, due to the fact that PBel is based on bilatices. For example, the UPF does not provide a direct operator which corresponds to the join operator on the trust lattice of PBel.

SMP

SMP is a logic for state-modifying authorisation policies [BN10]. The logic is based on Datalog, extended with statements allowing state modifications. It also provides a powerful proof system for reasoning about policies and sequences of access requests. Its ability to support state modifications results in a large similarity to the UPF. However, its main purpose, the underlying logic, and the applicability is quite different from the UPF. Interestingly however, one of the main case studies of the logic also considers of some policies of the NPfIT [Bec07]. However, the models were not meant to be applied for testing.

XACML

The eXtensible Access Control Markup Language (XACML) [OAS05] is an XML-based policy language with powerful policy combination operators. The standard defines both a declarative policy language, as well as a processing model describing how authorisation requests should be evaluated. Its main purpose is that policies coming from different sources can easily be integrated. With XACML we share the view that policies should

be modelled in a modular way and the focus should be on techniques for combination. While XACML is focused on the enforcement of policies, the focus of the UPF is on conformance testing. Furthermore, the composition operators we provide are strictly less general than the ones of the XACML. Having a smaller number of core operators leads to a reduced complexity. However, the most commonly used operators of XACML are also available in the UPF.

GPF

The *General Policy Framework (GPF)* [BL11,Bar09] is a policy framework for modelling a wide range of different kinds of policies. With the UPF we share the observation that a broad range of access control models can be reduced to a surprisingly small number of primitives together with a set of combinators or relations to build more complex policies. We also share the vision that the semantics of access control models should be formally defined. In contrast to the GPF, the UPF uses higher-order constructs and, more importantly, is geared towards machine support for (formally) transforming policies and supporting model-based test case generation approaches.

7.2 Model-based Testing

There are many model-based testing techniques available. Here, we ignore program testing tools like Java Pathfinder, PeX, or Korat, since their test methods are based on concrete programming code; Rather, we focus on methods to generate tests from a program-independent system model. As such, test generation techniques from such models are fairly well-known. Notable tools in the area of automated model-based testing include Microsoft's SpecExplorer [VCG+08], TorX [TB03], and TGV [JJ05]. Those tools are based on some conformance relation between a model of the system and the real system (for example based on ioco [vdBRT03]). Most of these tools are generic, but none is geared towards security policies.

In particular, one variant of TGV is close to some parts of our methodologies in that it uses Symbolic Finite State Machines for tests [CJMR07]. States are represented as formulae describing *sets of states*, and transitions can be guarded by constraints.

The Testing and Test Control Notation (TTCN-3) [SGVGD08] is a widely established test technology traditionally used in the telecommunication domain, which today has an even wider applicability. It is applied not only for testing the conformance and interoperability of communication protocols but also for testing the functionality, interoperation and performance of software-based systems in general. This approach offers local conformance tests of network components but has some difficulties in handling non-determinism.

7.3 Policy Conformance Testing

As regards to policy testing, the closest related works are those of [MX07a, HMHX07] presenting a test case generation approach based on change-impact analysis and mutation testing for a subset of XACML [OAS05] and that of [HA08] presenting a conformance testing approach for RBAC models using SAT solving techniques. [TMB07] present a conformance testing approach that generates test cases for checking that an OrBAC [KBM+03] policy (an extension of RBAC also modelling organisational contexts) complies with high-level compliance goals. Finally, there are several approaches, e.g. [PMLT08, HKX08], applying combinatorial testing to RBAC models. All the latter approaches have in common that they consider only RBAC models or simple extensions thereof. There do also exist some policy conformance testing approaches for other single domains, as for example for firewall policies (see below), or for privacy policies (see e.g. [SK08a]). In contrast, one of the main goals of our framework was to provide one uniform technique for a wide range of different policies.

7.4 Transformation Procedures

Our concept of policy transformations is an instantiation of the more general notion of *testability transformations*. [DKSC99] have already introduced a notion of testability for communication protocols using a specification based on finite state machines. They present a method for designing communication protocols that are "easy" to test. The closest related work however, is that of [HHH+04], which is further developed in [HHF05, Har08, HBB+08]. The authors define the concept of *testability transformations* as source-to-source transformations of programs that increase their testability. Similar to our work, the goal of these transformations is minimisation of the test cases required to achieve full test coverage (with respect to a given test adequacy criterion). While their work is based on transformation of the source code, that is of the system under test, we transform a formal specification, which makes our approach applicable in black-box testing scenarios. Furthermore, in contrast to their work, our transformations procedures allow for explicit modifications of the test adequacy criterion with respect to the original form of the specification; something which we believe to be necessary for being able to realistically perform conformance testing of large-scale security policies.

Moreover, we provide a uniform, tool-supported approach for formally analysing and transforming a test specification, and generating test cases for an implementation. And finally, our approach is based on a formal proof that the applied testability transformation preserves the semantics. These proofs are performed in the same framework used for test case generation. We believe these correctness proofs to be crucial for the method to be of value in a formal setting.

In contrast, transforming security policies, for example ones based on XACML [OAS05],

to improve their overall runtime performance is a well-known technique [Mis08, LTM08]. This approach also been extended to other policy languages. In particular, [LTM08] present algorithms based on decision-diagrams that minimise the number of rules of a firewall policy. We are not aware of any attempt to formally verify the correctness of these policy transformation techniques.

7.5 Modelling and Testing Firewall Policies

Firewall testing is a widespread research topic which reflects their importance in modern security infrastructures. Whereas we focus on conformance testing, many research approaches are focused on vulnerability and policy-independent implementation testing. Surprisingly, we did not find any work on random testing the conformance of a firewall with respect to a policy; this is in stark contrast to software testing where random testing has gained a certain popularity.

Several approaches for specification-based testing of firewalls have been proposed. For example, [EAIHAS05, EASW+07] present a policy segmentation technique where a policy is represented as a tree. They also give some measurements to the segments such that important segments can be tested more rigorously. They also present a policy generation technique. [JW01] propose the specification-based testing of firewalls which employs a formal model of the network and automatically derives test cases. It does however not really describe in which way these test cases are generated. Furthermore, it does not support an ordering of the firewall rules. [BFN+06] describes a formal model of protocols in HOL. Their level of abstraction is however much lower than ours and is therefore less suited for testing of policy conformance. [MK06] propose a policy-based host classification which can be used to detect errors and anomalies in a firewall policy. Senn et al. [SBC05, vB07] propose a language for specifying firewall policies and a framework for testing firewalls at the implementation level. While the framework includes tools for generating test cases with respect to a protocol specification, it lacks support for test case generation based on policies, allowing for conformance testing.

7.6 Modelling and Testing Health Care Policies

While there is a large body of literature adapting access control models to the specific needs of particular health care systems, e.g. [LLT00, HW04, WFSM02, RE06, Bec07, Bec05, EBM06, AdVF+10], only a few (i.e.. [Bec07, Bec05, EBM06]) discuss the particular needs of the NHS in England.

On the modelling part, the closest related work is that of Becker [Bec07, Bec05], presenting a formal model of the Information Governance policy using the authorisation

policy language Cassandra [BS04] and SMP. While his model covers more details than ours, such as the management of credentials, compared with our model it lacks a modular organisation. Moreover, the focus of this work is on the (efficient) enforcement of policies and not on the generation of test cases for validating compliance of an implementation. Overall, we agree with Becker that the requirements of the NHS cannot be modelled directly in traditional access control frameworks such as RBAC [SBM99] or Bell-LaPadula [BL96]. Finally, [EBM06] implemented an NHS care record service that supports the runtime enforcement of the RBAC-based sub-policies.

8 Conclusion and Future Work

8.1 Conclusion

In this thesis, we have presented the results of our work on a framework for the model-based conformance testing of security policies. We believe we have hereby advanced the state of the art considerably. For the first time, to the best of our knowledge, there exists a complete and generic framework for the model-based testing of a wide range of different kinds of security policies, integrating the complete testing process. The whole framework is based on formally rigorous principles, allowing for a tight integration with verification techniques and providing the means to formally reason about policy transformations, leading to a substantial improvement in efficiency of the testing process.

In more detail, one major part of the thesis is the development of the Unified Policy Framework (UPF). This modelling framework allows one to model a wide range of security policies, ranging from traditional access-control policies like RBAC, to very complex real-world policies as encountered in the NPfIT scenario, over to policies for firewalls and networks. This might come as a surprise, as especially the last example is traditionally considered quite remote from for example RBAC policies. With our framework we provided evidence that this is not the case, and that such policies are conceptually very similar.

Next to genericity, another main goal of the UPF was flexibility, in order to accommodate for real-world policies that are often only loosely inspired by traditional access control models instead of implementing them completely. This was of great importance in the NPfIT case study, which does not follow a single standard access control model. This is an instance of a more general lesson: real-world applications tend to be loosely inspired by abstract frameworks rather than to implement them faithfully.

A further design goal of the UPF was the focus on modularity, together with ease of combination of several subpolicies and dynamic transition behaviour. Again, a good example of the advantage of this approach can be seen in the NPfIT case study. It would be infeasible to model such a complex policy, together with the relevant system behaviour, without such support. However, this is also useful in less complex cases, as for example in the case of a firewall policy performing network address translation in addition to simple packet filtering behaviour.

Additionally, we have shown the practical relevance of the modelling framework by providing a methodology to use the policy models for model-based conformance testing using the test generation system HOL-TestGen. We support two kinds of testing approaches: unit tests are used to validate that a system implements a static policy correctly, while sequence tests ensure that the dynamic behaviour of a policy is implemented correctly. In the latter, informal security requirements can be translated into formal test specifications to be processed to test sequences for a possibly distributed system.

There exist a wide variety of different model-based testing techniques. While they share many common characteristics, they differ greatly in their internal methodologies. In [MUAPBL06], the authors have proposed a taxonomy of the different conceptual approaches of model-based testing. Our approach to model-based testing fits into this taxonomy as follows:

- Concerning the *model*, we allow both the specification of the system and of the environment, but only deal with discrete systems and do not treat any timing issues. The notation used for the models is based on Isabelle/HOL.
- Concerning the *test selection criterion*, we provide a combination of structural model coverage criteria, a data coverage criteria and ad-hoc test case specification. The technology used for generating the test cases is based on a theorem prover and explained in more detail in section 2.3.
- *Test execution* as presented in this thesis is offline only.

A common issue experienced throughout the work performed for this thesis was scalability. We have partially solved this by a technique called policy transformations, where a model of a policy is rewritten to a semantically equivalent model that is better suited for test case generation. These transformations can be proven to be correct directly inside the framework. In the firewall case study we provide a significant example of this approach, which has allowed us to achieve an efficiency gain making testing of industrial-size policies feasible. Such testability transformations are commonly used in testing. [Har08] discusses a list of open problems in testability transformations. With our work, we make a significant contribution to the final problem in his list: testability transformations for specification-based testing.

In this thesis, we have presented two large-scale case studies. The first one consists of modelling the complex information governance policies of the National Programme for Information Technology (NPfIT) in the English NHS. By doing this, we have provided evidence of the strength of the modelling framework. While the concrete policies modelled can be considered as quite specific for this single scenario, we believe that many of the findings can be transferred to other application domains. The policy concepts themselves are very interesting and can be of use elsewhere. We have also used these models for generation of test cases and provide support for automatic testing of web-service

based applications.

The second case study is in the domain of firewalls and networks. We present how the UPF can be used to model different kinds of firewall policies and how to combine them. Furthermore, we present a significant policy transformation procedure, and outline the test case generation process including its empirical evaluation for a large number of scenarios. Finally, we presents a test execution tool for firewalls. In this setup, we believe we have developed the first domain-specific test tool to integrate the specification, formal analysis, transformation, and test case generation of firewall policies.

8.2 Future Work

In this section, we present some directions for future work. Regarding the modelling framework, we plan to apply it to other access control models. Furthermore there remain a large number of possibilities which make the framework usable for engineers not proficient in formal methods. One possibility is to develop abstractions in terms of reusable patterns and templates from successful models, and integrate them into mainstream software development environments, such as Eclipse, to benefit from the productivity aids already available. It should be possible to define a small number of templates that cater for the majority of common policy scenarios, in which other aspects of the test specification are held for example as plain text.

Regarding test case generation, we plan to develop specific test case generation algorithms. Such domain-specific algorithms allow, by exploiting knowledge about the structure of access control models, respectively the UPF, for a deeper exploration of the test space possibly resulting in an improved test coverage. In fact, the normalisation transformation for firewall policies, carried out prior to the usual test case generation, is already one example of such a domain-specific test case generation algorithm. Such domain-specific algorithms do not necessarily have to based on Isabelle/HOL and HOL-TestGen.

In general, the policy conformance testing setting is mostly black-box. However, there might be cases where the systems to be tested against are provided by their source code. For those cases, it is appealing to explore the possibilities to combine our black-box based testing approach with white-box approaches. This would allow for a good code coverage in addition to specification coverage. Such approaches are called *gray-box* testing. Considerable work in this areas has been performed by [KGTB07], but not particularly in the domain of security policies.

With regards to policy transformation techniques, further transformation procedures for standard access control models and for generic UPF policies are needed. Furthermore, developing formal testability transformations for sequence testing seems to be particularly appealing. This work would address the second to last problem of [Har08]:

testability transformations for specifications based on finite-state machines. Moreover, considering testability transformations that do not preserve the semantics of the specifications, as suggested by [HBB+08], requires the development of new test hypotheses. Here, the integrated approach of HOL-TestGen allows for the formal computation and analysis of such hypotheses (cf. [BBW08b]). Finally, the relation between testability transformations and fault models needs to be investigated further.

More work could be performed regarding evaluation of the framework with respect to the ability of finding typical failures. While we have applied our framework to several firewall configurations and prototypical NPfIT applications, more empirical evaluation gained through application of the framework to further scenarios within those and other domains would be beneficial. Another promising approach is to apply the framework to scenarios for which there exist domain-specific testing frameworks and compare the quality of the test cases generated by these frameworks with the ones generated by the UPF.

Bibliography

[AdVF+10] Claudio A. Ardagna, Sabrina De Capitani di Vimercati, Sara Foresti, Tyrone W. Grandison, Sushil Jajodia, and Pierangela Samarati. Access control for smarter healthcare using policy spaces. *Computers & Security*, 2010.

[And02] Peter B. Andrews. *Introduction to Mathematical Logic and Type Theory: To Truth through Proof*. Kluwer Academic Publishers, 2nd edition, 2002.

[AOH03] Paul Ammann, A. Jefferson Offutt, and Hong Huang. Coverage Criteria for Logical Expressions. In *ISSRE*, pages 99–107. IEEE Computer Society, 2003.

[Bar09] Steve Barker. The next 700 access control models or a unifying meta-model? In *SACMAT*, pages 187–196. ACM Press, 2009.

[BBF01] Elisa Bertino, Piero Andrea Bonatti, and Elena Ferrari. TRBAC: A temporal role-based access control model. *ACM Trans. Inf. Syst. Secur.*, 4(3):191–233, 2001.

[BBKW10] Achim D. Brucker, Lukas Brügger, Paul Kearney, and Burkhart Wolff. Verified Firewall Policy Transformations for Test-Case Generation. In *International Conference on Software Testing, Verification, and Validation (ICST)*. IEEE Computer Society, 2010.

[BBKW11] Achim D. Brucker, Lukas Brügger, Paul Kearney, and Burkhart Wolff. An Approach to Modular and Testable Security Models of Real-world Health-care Applications. In *ACM symposium on access control models and technologies (SACMAT)*, pages 133–142. ACM Press, 2011.

[BBW08a] Achim D. Brucker, Lukas Brügger, and Burkhart Wolff. Model-based Firewall Conformance Testing. In Kenji Suzuki and Teruo Higashino, editors, *Testcom/FATES 2008*, number 5047 in LNCS, pages 103–118. Springer-Verlag, 2008.

[BBW08b] Achim D. Brucker, Lukas Brügger, and Burkhart Wolff. Verifying Test-Hypotheses: An Experiment in Test and Proof. *Electronic Notes in Theoretical Computer Science*, 220(1):15–27, 2008.

[BDMN06] Adam Barth, Anupam Datta, John C. Mitchell, and Helen Nissenbaum. Privacy and Contextual Integrity: Framework and Applications. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 184–198, Washington, DC, USA, 2006. IEEE Computer Society.

[Bec05] Moritz Y. Becker. A formal security policy for an NHS electronic health record service. Technical Report UCAM-CL-TR-628, University of Cambridge, 2005.

[Bec07] Moritz Y. Becker. Information governance in NHS's NPfIT: A case for policy specification. *International Journal of Medical Informatics*, 2007.

[Bel77] Nuel D Belnap. A useful four-valued logic. In G Epstein and Editors Dunn, J, editors, *Modern Uses of MultipleValued Logic*, pages 8–37. Reidel, 1977.

[BFN+06] Steve Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. Engineering with logic: HOL specification and symbolic-evaluation testing for TCP implementations. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *POPL*, pages 55–66. ACM Press, 2006.

[BH11] Glenn Bruns and Michael Huth. Access control via belnap logic: Intuitive, expressive, and analyzable policy composition. *ACM Trans. Inf. Syst. Secur.*, 14:9:1–9:27, June 2011.

[Bib77] Biba, J. K. Integrity Considerations for Secure Computer Systems, 1977.

[BL96] D. Elliott Bell and Leonard J. LaPadula. Secure Computer Systems: A Mathematical Model, Volume II. In *Journal of Computer Security 4*, pages 229–263, 1996. An electronic reconstruction of *Secure Computer Systems: Mathematical Foundations*, 1973.

[BL11] Steve Barker and Gillian Lowen. GPF: A General Policy Framework. In *POLICY*, pages 117–120. IEEE Computer Society, 2011.

[BN10] Moritz Y. Becker and Sebastian Nanz. A Logic for State-Modifying Authorization Policies. *ACM Transactions on Information and System Security*, 13(3):1–28, 2010.

[BP09] Achim D. Brucker and Helmut Petritsch. Extending access control models with break-glass. In Barbara Carminati and James Joshi, editors, *ACM symposium on access control models and technologies (SACMAT)*, pages 197–206. ACM Press, New York, NY, USA, 2009.

[BS04] Moritz Y. Becker. and Peter Sewell. Cassandra: flexible trust management, applied to electronic health records. In *CSF*, pages 139–154. IEEE

Computer Society, 2004.

[But74] Butler W. Lampson. Protection. *Operating Systems Review*, 8(1):18–24, 1974.

[BW07] Achim D. Brucker and Burkhart Wolff. Test-Sequence Generation with HOL-TestGen – With an Application to Firewall Testing. In Bertrand Meyer and Yuri Gurevich, editors, *TAP 2007: Tests And Proofs*, number 4454 in LNCS, pages 149–168. Springer-Verlag, 2007.

[BW12] Achim D. Brucker and Burkhart Wolff. On Theorem Prover-based Testing. *Formal Aspects of Computing*, 2012. To appear.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, 1940.

[cis08] Securing Cyberspace for the 44th Presidency. Technical report, Center for Strategic and International Studies (CSIS), 2008.

[CJMR07] Camille Constant, Thierry Jeron, Herve Marchand, and Vlad Rusu. Integrating formal verification and conformance testing for reactive systems. *IEEE Transactions on Software Engineering*, 33:558–574, 2007.

[CLMR02] Lorrie Cranor, Marc Langheinrich, Massimo Marchiori, and Joseph Reagle. The Platform for Privacy Preferences 1.0 (P3P1.0) Specification. W3C Recommendation, April 2002.

[Dep03] Department of Health. Confidentiality. Code of Practice, 2003.

[Dep10] Department of Health. Spine: Frequently Asked Questions, 2010. http://www.connectingforhealth.nhs.uk/systemsandservices/spine/faqs.

[DF93] Jeremy Dick and Alain Faivre. Automating the Generation and Sequencing of Test Cases from Model-Based Specifications. In J.C.P. Woodcock and P.G. Larsen, editors, *Formal Methods Europe 93: Industrial-Strength Formal Methods*, volume 670 of *LNCS*, pages 268–284. Springer-Verlag, April 1993.

[Dij72] Edsger W. Dijkstra. Structured programming. chapter i: Notes on structured programming. pages 1–82. Academic Press Ltd., London, UK, UK, 1972.

[DKSC99] R. Dssouli, K. Karoui, K. Saleh, and O. Cherkaoui. Communications software design for testability: specification transformations and testability measures. *Information and Software Technology*, 41(11-12):729–743, 1999.

[dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[dVSJ05] Sabrina De Capitani di Vimercati, Pierangela Samarati, and Sushil Jajodia. Policies, models, and languages for access control. In Subhash Bhalla, editor, *DNIS*, volume 3433 of *Lecture Notes in Computer Science*, pages 225–237. Springer, 2005.

[EAIHAS05] Adel El-Atawy, K. Ibrahim, H. Hamed, and Ehab Al-Shaer. Policy segmentation for intelligent firewall testing. In *NPSec 05*, pages 67–72. IEEE Computer Society, 2005.

[EASW+07] Adel El-Atawy, Taghrid Samak, Zein Wali, Ehab Al-Shaer, Frank Lin, Christopher Pham, and Sheng Li. An Automated Framework for Validating Firewall Policy Enforcement. In *Policy '07*, pages 151–160. IEEE Computer Society, 2007.

[EBM06] D.M. Eyers, J. Bacon, and Ken Moody. OASIS role-based access control for electronic health records. In *IEE Software*, volume 153, pages 16–23. IEE, 2006.

[EFW01] Ibrahim K. El-Far and James A. Whittaker. Model-based Software Testing. *Encyclopedia on Software Engineering*, 2001.

[Ejs11] Simon Ejsing. Application of Code Coverage to Model Based Testing. http://appmbt.blogspot.com/2011/05/application-of-code-coverage-to-model.html, 2011. Blog: Applied Model Based Testing - Experiences & Examples.

[Gau95] M.-C. Gaudel. Testing can be formal, too. In *TAPSOFT'95, International Joint Conference, Theory And Practice of Software Development*, volume 915 of *Lecture Notes in Computer Science*, pages 82–96, Aarhus, Denmark, 1995. Springer Verlag.

[GF05] Pedro Gama and Paulo Ferreira. Obligation policies: An enforcement platform. In *Proceedings of the Sixth IEEE International Workshop on Policies for Distributed Systems and Networks*, POLICY '05, pages 203–212, Washington, DC, USA, 2005. IEEE Computer Society.

[GG75] John B. Goodenough and Susan L. Gerhart. Toward a theory of test data selection. In *Proceedings of the international conference on Reliable software*, pages 493–510, 1975.

[GKM+91] Anne Geraci, Freny Katki, Louise McMonegal, Bennett Meyer, and Hugh

Porteous. *IEEE Standard Computer Dictionary. A Compilation of IEEE Standard Computer Glossaries.* pub-ieee, 1991.

[HA08] Hongxin Hu and Gail-Joon Ahn. Enabling verification and conformance testing for access control model. In *SACMAT*, pages 195–204. ACM Press, 2008.

[Har08] M. Harman. Open Problems in Testability Transformation. In *Software Testing Verification and Validation Workshop (*ICSTW*)*, pages 196–209, 2008.

[HBB+08] Mark Harman, André Baresel, David Binkley, Robert M. Hierons, Lin Hu, Bogdan Korel, Phil McMinn, and Marc Roper. Testability Transformation - Program Transformation to Improve Testability. In Robert M. Hierons, Jonathan P. Bowen, and Mark Harman, editors, *Formal Methods and Testing*, volume 4949 of *LNCS*, pages 320–344. Springer-Verlag, 2008.

[HHF05] Robert M. Hierons, Mark Harman, and Chris Fox. Branch-Coverage Testability Transformation for Unstructured Programs. *Comput. J.*, 48(4):421–436, 2005.

[HHH+04] Mark Harman, Lin Hu, Rob Hierons, Joachim Wegener, Harmen Sthamer, André Baresel, and Marc Roper. Testability Transformation. *IEEE Trans. Softw. Eng.*, 30(1):3–16, 2004.

[HKX08] V.C. Hu, D.R. Kuhn, and Tao Xie. Property Verification for Generic Access Control Models. In *EUC*, volume 2, pages 243–250, 2008.

[HMHX07] V.C. Hu, E. Martin, JeeHyun Hwang, and Tao Xie. Conformance Checking of Access Control Policies Specified in XACML. In *COMPSAC*, volume 2, pages 275–280, 2007.

[HRU76] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. Protection in Operating Systems. *Commun. ACM*, 19(8):461–471, 1976.

[HW04] Junzhe Hu and Alfred C. Weaver. Dynamic, Context-Aware Access Control for Distributed Healthcare Applications. In *PSPT*, 2004.

[HXCL08] JeeHyun Hwang, Tao Xie, Fei Chen, and A.X. Liu. Systematic structural testing of firewall policies. In *Reliable Distributed Systems, 2008. SRDS '08. IEEE Symposium on*, pages 105 –114, oct. 2008.

[JJ05] Claude Jard and Thierry Jeron. Tgv: theory, principles and algorithms: A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *Int. J. Softw. Tools Technol. Transf.*, 7(4):297–315, 2005.

[JW01] Jan Jürjens and Guido Wimmel. Specification-Based Testing of Firewalls. In Dines Bjørner, Manfred Broy, and Alexandre V. Zamulin, editors, *Ershov Memorial Conference*, volume 2244 of *LNCS*, pages 308–316. Springer-Verlag, 2001.

[KBM+03] Anas Abou El Kalam, Salem Benferhat, Alexandre Miège, Rania El Baida, Frédéric Cuppens, Claire Saurel, Philippe Balbiani, Yves Deswarte, and Gilles Trouessin. Organization based access control. In *POLICY*, pages 120–131. IEEE Computer Society, 2003.

[KGTB07] Nicolas Kicillof, Wolfgang Grieskamp, Nikolai Tillmann, and Victor Braberman. Achieving both model and code coverage with automated gray-box testing. In *Proceedings of the 3rd international workshop on Advances in model-based testing*, A-MOST '07, pages 1–11, New York, NY, USA, 2007. ACM.

[KS02] Günter Karjoth and Matthias Schunter. A Privacy Policy Model for Enterprises. In *CSFW*, pages 271–281. IEEE Computer Society, 2002.

[LLT00] J.J. Longstaff, M.A. Lockyer, and M.G. Thick. A Model of Accountability, Confidentiality and Override for Healthcare and other Applications. In *Role-based access control*, pages 71–76. ACM Press, 2000.

[LTM08] Alex X. Liu, Eric Torng, and Chad Meiners. Firewall Compressor: An Algorithm for Minimizing Firewall Policies. In *IEEE Conference on Computer Communications (Infocom)*, 2008.

[MA01] M.J. Moyer and M. Abamad. Generalized role-based access control. *Distributed Computing Systems, 2001. 21st International Conference on.*, pages 391–398, Apr 2001.

[MCMD11] Srdjan Marinovic, Robert Craven, Jiefei Ma, and Naranker Dulay. Rumpole: a flexible break-glass access control model. In *Proceedings of the 16th ACM symposium on Access control models and technologies*, SACMAT '11, pages 73–82, New York, NY, USA, 2011. ACM.

[Mis08] Philip Miseldine. Automated XACML policy reconfiguration for evaluation optimisation. In Bart De Win, Seok-Won Lee, and Mattia Monga, editors, *Software Engineering for Secure Systems (*SESS*)*, pages 1–8. ACM Press, 2008.

[MK06] Robert Marmorstein and Phil Kearns. Firewall analysis with policy-based host classification. In LISA, pages 4–4. Usenix Association, 2006.

[Mog91] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[MUAPBL06] Mark Utting, Alexander Pretschner, and Bruno Legeard. A Taxonomy of Model-Based Testing, 2006.

[MX07a] E. Martin and Tao Xie. Automated Test Generation for Access Control Policies via Change-Impact Analysis. In *SESS*, pages 5–5, 2007.

[MX07b] Evan Martin and Tao Xie. A fault model and mutation testing of access control policies. In *World Wide Web (WWW)*, pages 667–676. ACM, 2007.

[NHS06] NHS, Malcolm Oswald. Checklist for LSP's Implementing Legitimate Relationships, 2006. restricted commercial.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL—A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[OAS05] extensible Access Control Markup language (XACML), Version 2.0, 2005.

[Ola97] Olaf Müller and Konrad Slind. Treating Partiality in a Logic of Total Functions. *Comput. J.*, 40(10):640–652, 1997.

[PMLT08] A. Pretschner, T. Mouelhi, and Y. Le Traon. Model-Based Tests for Access Control Policies. In *ICST*, pages 338–347. IEEE Computer Society, 2008.

[RBA] *American National Standard for Information Technology – Role Based Access Control*. INCITS 359-2004.

[RE06] Lillian Rostad and Ole Edsberg. A Study of Access Control Requirements for Healthcare Systems Based on Audit Trails from Access Logs. In *ACSAC*, pages 175–186. IEEE Computer Society, 2006.

[SB07] Christoph Sprenger and David Basin. A monad-based modeling and verification toolbox with application to security protocols. In K. Schneider and J. Brandt, editors, *TPHOLs*, volume 4732 of *LNCS*, pages 301–317. Springer, 2007.

[SBC05] Diana Senn, David Basin, and Germano Caronni. Firewall Conformance Testing. In Ferhat Khendek and Rachida Dssouli, editors, *TestCom 2005*, volume 3502 of *LNCS*, pages 226–241. Springer-Verlag, May 2005.

[SBM99] Ravi Sandhu, Venkata Bhamidipati, and Qamar Munawer. The ARBAC97 model for role-based administration of roles. *ACM TISSEC*, 2(1):105–135, 1999.

[SCFY96] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E.

Youman. Role-Based Access Control Models. *Computer*, 29:38–47, February 1996.

[SGVGD08] Ina Schieferdecker, Jens Grabowski, Theofanis Vassiliou-Gioles, and George Din. *The Test Technology TTCN-3*. Springer-Verlag, Berlin Heidelberg, April 2008.

[SK08a] Percy Antonio Pari Salas and Padmanabhan Krishnan. Testing Privacy Policies Using Models. In Antonio Cerone and Stefan Gruner, editors, *SEFM*, pages 117–126. IEEE Computer Society, 2008.

[SK08b] P.P. Salas and P. Krishnan. Testing privacy policies using models. In *Software Engineering and Formal Methods, 2008. SEFM '08. Sixth IEEE International Conference on*, pages 117 –126, nov. 2008.

[Spi92] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, Inc., 2nd edition, 1992.

[SV01] Pierangela Samarati and Sabrina De Capitani di Vimercati. Access Control: Policies, Models, and Mechanisms. In *Revised versions of lectures given during the IFIP WG 1.7 International School on Foundations of Security Analysis and Design on Foundations of Security Analysis and Design: Tutorial Lectures*, FOSAD '00, pages 137–196, London, UK, 2001. Springer-Verlag.

[TB03] G. J. Tretmans and H. Brinksma. Torx: Automated model-based testing. In A. Hartman and K. Dussa-Ziegler, editors, *First European Conference on Model-Driven Software Engineering, Nuremberg, Germany*, pages 31–43, December 2003.

[The97] The Caldicott Committee. Report on the Review of Patient-Identifiable Information, 1997.

[TK06] Michael Carl Tschantz and Shriram Krishnamurthi. Towards reasonability properties for access-control policy languages. In *Proceedings of the eleventh ACM symposium on Access control models and technologies*, SACMAT '06, pages 160–169, New York, NY, USA, 2006. ACM.

[TMB07] Yves Le Traon, Tejeddine Mouelhi, and Benoit Baudry. Testing Security Policies: Going Beyond Functional Testing. In *ISSRE*, pages 93–102, 2007.

[Tre08] Jan Tretmans. Model based testing with labelled transition systems. In Robert Hierons, Jonathan Bowen, and Mark Harman, editors, *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, pages 1–38. Springer Berlin / Heidelberg, 2008.

[UL06] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[vB07] Diana von Bidder. *Specification-based Firewall Testing*. Ph.d. thesis, ETH Zurich, 2007. ETH Dissertation No. 17172. Diana von Bidder's maiden name is Diana Senn.

[VCG+08] Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson. Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer. In Robert M. Hierons, Jonathan P. Bowen, and Mark Harman, editors, *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, pages 39–76. Springer, 2008.

[vdBRT03] Machiel van der Bijl, Arend Rensink, and Jan Tretmans. Compositional testing with ioco. In *FATES*, pages 86–100, 2003.

[Wen02] Markus M. Wenzel. *Isabelle/Isar — a versatile environment for human-readable formal proof documents*. PhD thesis, TU München, München, February 2002.

[WFSM02] Marc Wilikens, Simone Feriti, Alberto Sanna, and Marcelo Masera. A context-related authorization and access control method based on RBAC: A case study from the health care domain. In *SACMAT*, pages 117–124. ACM Press, 2002.

[WL93] Thomas Y. C. Woo and Simon S. Lam. Authorizations in Distributed Systems: A New Approach. *Journal of Computer Security*, 2(2-3):107–136, 1993.

[ZHM97] Hong Zhu, Patrick A.V. Hall, and John H. R. May. Software unit Test Coverage and Adequacy. *ACM Computing Surveys*, 29(4):366–427, December 1997.

Glossary

ARBAC Administrative Role-based Access Control.

CRS Care Records Service.

DMZ demilitarized zone.

DNF disjunctive normal form.

FTP File Transfer Protocol.

GP General Practitioner.

HOL Higher-order Logic.

IG Information Governance.

IP Internet Protocol.

LR Legitimate Relationship.

NAT Network Address Translation.

NPfIT National Programme for Information Technology.

NRD National RBAC Database.

PAT Port Address Translation.

PC Patient's Consent.

PDS Personal Demographic Service.

PSIS Personal Spine Information Service.

RBAC Role-based Access Control.

SCR Summary Care Record.

SE Sealed Envelope.

SML Standard Meta Language.

SMT Satisfiability Modulo Theories.

UPF Unified Policy Framework.

URP User Role Profile.

VoIP Voice over IP.

XACML eXtensible Access Control Markup Language.

Curriculum Vitae

Personal Information

Name:	**Lukas Brügger**
Citizen of:	Plaffeien FR
Date of Birth:	14 June 1982

Education

9/07–6/12	**ETH Zurich, Switzerland** doctoral student in the Institute of Information Security headed by Prof. David Basin
10/02–8/07	**ETH Zurich, Switzerland** MSc. degree in Computer Science MSc. Thesis: *Proof-Support for IMP++ in HOL-OCL. A Typed Verification Approach for Object-oriented Programs*
9/97–6/01	**Kollegium Heilig-Kreuz, Freiburg, Switzerland** Matura Typ C

Work Experience

9/07–3/12	Teaching Assistant for various courses at ETH Zurich
9/07–12/11	Research Project: Model-based Testing of Security Policies, with BT group plc and University Paris-Sud
9/06–12/06	Internship at BT group plc (UK) in Model-driven Security
4/05–6/05	Student Research Assistant at ETH Zurich