An Approach to Modular and Testable Security Models of Real-world Health-care Applications

Achim D. Brucker
SAP Research
Vincenz-Priessnitz-Str. 1
76131 Karlsruhe, Germany
achim.brucker@sap.com

Lukas Brügger[*]
Information Security
ETH Zurich
8092 Zurich, Switzerland
lukas.bruegger@inf.ethz.ch

Paul Kearney
Security Futures Practice
BT Innovate & Design
Adastral Park, Ipswich, UK
paul.3.kearney@bt.com

Burkhart Wolff[†]
Université Paris-Sud
Parc Club Orsay Université
91893 Orsay Cedex, France
wolff@lri.fr

ABSTRACT

We present a generic modular policy modelling framework and instantiate it with a substantial case study for model-based testing of some key security mechanisms of applications and services of the NPfIT. NPfIT, the National Programme for IT, is a very large-scale development project aiming to modernise the IT infrastructure of the National Health Service (NHS) in England. Consisting of heterogeneous and distributed applications, it is an ideal target for model-based testing techniques of a large system exhibiting critical security features.

We model the four *information governance principles*, comprising a role-based access control model, as well as policy rules governing the concepts of patient consent, sealed envelopes and legitimate relationships. The model is given in Higher-order Logic (HOL) and processed together with suitable test specifications in the HOL-TESTGEN system, that generates test sequences according to them. Particular emphasis is put on the modular description of security policies and their generic combination and its consequences for model-based testing.

Categories and Subject Descriptors

D.4.6 [**Software**]: Operating Systems—*Security and Protection*

General Terms

Security, Languages

[*]This work was partly supported by BT Group plc.
[†]This work was partly supported by the Digiteo Foundation.

Keywords

Security testing, model-based testing, electronic health-care record, NPfIT

1. INTRODUCTION

The National Health Service (NHS) in England's National Programme for Information Technology (NPfIT) is one of the most ambitious and challenging ongoing IT projects worldwide [9].[1] At the heart of the project lies the development of a nationwide on-line service, also known as the *Spine*, enabling health professionals and patients to access electronic health records. Many lessons can be learnt from studying the security mechanisms of a real-world system—perhaps most notably, that the traditional borderline between *security* and *safety*, (where security is understood as "protecting the confidentiality, integrity and availability of information systems" and safety is viewed as "protection of health and life of humans"), becomes pretty artificial, since erroneous or missing health-care records can lead to serious harm to patients [2, 10]. In contrast to such "compartmentalised thinking," the analysis of real-world scenarios needs a more interdisciplinary approach combining formal methods, formal testing, computer security, and software engineering.

The security requirements—called Information Governance (IG) in the NPfIT terminology—are informally specified in many official documents (e. g. [17, 18]). Ensuring that all of the applications and services of the NPfIT individually and collectively comply with these policies, is a very difficult task. An approach based on verification is impossible to be successful in such a complex scenario with a heterogeneous and distributed code-base. Model-based Testing (MBT) is an ideal approach to address these problems and can help to increase confidence in the security mechanisms of the NPfIT. We advocate a technique to create a model of the relevant requirements, automatically generate test sequences from the model and run them against the real system to both validate our formalisation as well as finding bugs in the various implementations. The challenges of *modelling* these Information Governance principles are manifold:

[1]Possible future and recently announced changes in name and scope of the programme by the government are not considered in this paper.

- 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 Role-based Access Control (RBAC) [1] are insufficiently expressive to capture complex policies such as *Legitimate Relationships* (LR), *Sealed Envelopes* (SE), or *Patient Consent* (PC) management. Therefore, both a framework for their *uniform* modelling as well as their combination is needed.
- 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 [1, 30] models.
- The requirements are mandated by laws, official guidelines and ethical positions (e. g. [16, 32]) that are prone to change. Such changes have to be enforced throughout a distributed and heterogeneous system, i. e. the Spine and the applications that access it from various sites. Moreover, the accessing applications also have to conform to local policies.

This world of informal, high-level information governance descriptions is in contrast to the formalised world of policies in *Higher-order Logic* (HOL). We have developed a somewhat non-standard view on the fundamental concept of *policies*. This view has arisen from prior experience in the modelling of network (firewall) policies [13]. Instead of regarding policies as relations on resources, sets of permissions, etc., we emphasise the view that a policy is a *policy decision function* that grants or denies access to resources, permissions, etc. In other words, we model the concrete function that implements the policy decision point in a system. An advantage of this view is that it is compatible with many different policy models. Furthermore, this function is typically a large cascade of nested conditionals, using conditions referring to an internal state and to security contexts of the system or a user. This cascade of conditionals can easily be decomposed into a set of test cases similar to transformations used for Binary Decision Diagrams (BDDs). From the modelling perspective, our system uses HOL as its input language, offering all the expressive power of a functional programming language, including the possibility to define higher-order combinators. In more detail, we model policies as partial functions (written as $\_ \rightharpoonup \_$) based on input data of type α (e. g., arguments, system state, security context) to output data of type β:

$$\textbf{types}\ \alpha \mapsto \beta = \alpha \rightharpoonup \beta\ \mathsf{decision}\ ,$$

where the enumeration type β decision just consists of the two variants allow β and deny β. *Partial* functions are used since we describe elementary policies by partial system behaviour, which are glued together by operators such as function override and functional composition.

A particular instance of this generic concept of policy is the *transition policy*. Transition policies have the form:

$$\alpha \times \sigma \mapsto \beta \times \sigma$$

where σ refers to some system state. Transition policies are, as we will see later, isomorphic to state-exception monads and therefore amenable to the approach of HOL-TestGen [14] to sequence testing (based on models in HOL). Since policies in our sense—e. g. a decision function or decision table for an RBAC model—are possible elements of a system state σ, we can have policies that transform policies in our framework:

$$(\alpha \times (\gamma \mapsto \delta)) \mapsto (\beta \times (\gamma \mapsto \delta))$$

Since these constructs in HOL have a type of order two, we call this form of transition policies *second order policies.*

Our contributions are three-fold:

1) we present a modular modelling framework for security policies,
2) we instantiate our modelling framework with a large, real-world case study in the health-care domain,
3) we show how to generate test sequences for this case study using HOL-TestGen [11, 12]. The generated test sequences can be used for testing, e. g. Web Service-based, applications that access the Spine.

The organisation of the paper is as follows: in Section 2 we explain our background, i. e. the NPfIT and HOL-TestGen, in Section 3 we present briefly our framework for modelling security polices and in Section 4 its instantiation with the NPfIT concepts. In Section 5 we discuss the generation of test data on the basis of *test specifications*, i. e. concrete properties of the system.

2. BACKGROUND

2.1 Overall System Architecture of the NPfIT

At the core of the National Programme for Information Technology (NPfIT) is the *Spine*, a collection of services provided centrally and used by applications installed locally in hospitals, doctors practises, etc. The most important of these services is the *Care Records Service* (CRS), providing authorised health care staff (here called *users*) access to an electronic health record (called *Summary Care Record* (SCR)) for every *patient* in England.

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 Information Governance (IG) principles are expressed in terms of four main concepts: Role-based Access Control (RBAC), Legitimate Relationships (LR), Sealed Envelopes (SE) and Patient Consent (PC) Management.

2.1.1 Role-Based Access Control (RBAC)

NPfIT uses a variant of administrative RBAC [30] to control who can access what system functionality.[2] Each user is assigned one or more *User Role Profile* (URP). Each URP permits the user to perform several *activities*. Activities are generic descriptions of business functionality grouped in a hierarchy. Each application that is part of the NPfIT, must define its set of application functions which have to be controlled by RBAC. Each of these functions is mapped to one or more activities. Thus, each permitted activity gives a user

[2]In this paper, we follow the naming convention of the NPfIT documents which, in some cases, deviates from widely-used RBAC definitions such as [1, 30].

access to a set of application functions. A URP contains the following main elements:

Job Role: a generic description of the job of a user, e. g. General Practitioner (GP).

Areas of Work: e. g. mental health. Partly, in combination with a specific role, confer additional access rights.

Additional Activities: used to grant additional permissions to an individual user. This adds flexibility and minimises the number of roles required. They grant a user permissions which he otherwise would not have.

Thus, there are three reasons to grant a user rights to perform an activity: a) It is a generic requirement of a Job Role b) It is a requirement of members of a Job Role working in a specialised discipline c) It is an explicit requirement of the individuals job. The mappings for the first two points are common throughout the NPfIT and stored in a national database on the Spine, containing the set of activities granted for each role and role-area of work combination.

2.1.2 *Legitimate Relationships (LR)*

Whereas RBAC expresses the need to use application functions, Legitimate Relationships (LR) express the need to view or change records of particular patients. A user is only allowed to access the data of patients in whose care he is actually involved. Users are assigned to hierarchically ordered *workgroups* that reflect the organisational structure of a workplace.

There are ten different types of LRs. Eight of them are between a patient and a workgroup, two between a patient and a single user. LRs can either be active or frozen. Loosely speaking an active LR denotes a current relationship, whereas a frozen LR indicates a previous relationship. A frozen LR only allows access to data created before a specific time in the past. All LRs carry an expiry date. There is a complex set of rules governing the dynamic behaviour of these LRs. Sometimes, users can also self-claim an LR (e. g. in case of an emergency), but this always triggers a message to the responsible privacy officer (the Caldicott Guardian).

While the circumstances under which LRs can be created are clearly defined, this is done by each application itself, usually transparently to a user as part of the workflow within an application. For example, when a General Practitioner (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 of the LRs to the Spine, which stores all the current LRs together with the workgroup memberships and their hierarchy. Applications can query the Spine to find out if a user has an LR with a patient. The Spine itself does not enforce correct treatment of the LRs by the applications.

2.1.3 *Patient Consent (PC)*

Patients can opt out of having a Summary Care Record (SCR) at all, in which case no health care details will be uploaded to the Spine; a blank SCR will be created. If an SCR is created, the patient may choose to be asked every time a user attempts to add or read data and can refuse this. Alternatively, a patient can grant this right once and for all. If a patient with an SCR decides to opt out later, his care record will be suppressed. A complete deletion of a record is only possible in special cases.

Some of the data will always be shared, even if users dissent, e. g. demographic data and information about a patient's GP.

2.1.4 *Sealed Envelopes (SE)*

The sealing concept is used to hide parts of a health care record from users. There are three different kinds of seals:

Seal: Sealed information is hidden except from users in the same workgroup as the creator of the seal. Other users can detect sealed parts and unlock a seal in specific cases.

Seal and lock: Like *seal*, but without the possibility of detection of sealed parts.

Clinician seal: Used to hide data from a patient. Users are able to detect sealed parts and to unlock them, but are supposed to keep them confidential from the patient. A patient cannot detect parts sealed in this way when accessing his record directly.

Patients cannot create seals themselves; only users can do this on their behalf. Some data is sealed automatically, e. g. test results, while other data can never be sealed.

Not all of the concepts mentioned have been implemented yet, some are not yet completely specified and are subject to change. In particular, the concept of Sealed Envelopes has recently been dropped from the NPfIT. In our model, we therefore sometimes had to use simplified approximations in the spirit of the policies. In rare cases where we could not get access to the latest policy documents, we made our own assumptions or used earlier specifications. While we tried to stay as close as possible to the "real" policy, our model cannot be viewed as a true description of what is in place now or will be there in a final system implementation.

2.1.5 *Example*

As an example, consider a situation in which Alice is both a *clinical practitioner* and a *clerical*, Bob and John are *nurses*. Bob may add or remove people to or from workgroups. Alice, in her role as clinical practitioner, and John belong to the workgroup *surgery*. Alice, in her role as clerical, and Bob belong to the workgroup *orthopedics*. Now assume two patients Paula and Pablo whose patient records should be (partially) available to Alice, Bob, and John. Modelling such a scenario in traditional security models such as ANSI conform RBAC [1] or Bell-LaPadula [8] is at least difficult, if not impossible. In the NPfIT framework we can model dynamic relationships between patients and their medical practitioners. For example, we can model that Paula has a *legitimate relationship* with the working group *surgery*, Alice is his general practitioner. Her record has three entries. The first one is sealed (open) for *surgery*, the second one is sealed (open) for *orthopedics*, the third one is open. Pablo has a legitimate relationship with *orthopedics*, and no entries in his patient record.

2.2 A Note on HOL-TestGen

HOL-TESTGEN is an interactive, i. e. semi-automated, test tool for specification-based tests built upon Isabelle/HOL. Isabelle [26] is an interactive modelling and theorem proving environment; among other logics, Isabelle supports Higher-order Logic (HOL) [15]. HOL is a classical logic with equality, enriched by total higher-order functions; thus, it offers the usual logical connectives $\neg A$, $A \wedge B$, $A \rightarrow B$, $A = B$ and $\forall x.\ P(x)$, etc. HOL is a language typed with Hindley/Milner polymorphism; type variables are denoted by α, β, γ, etc. Function types are written by $\alpha \Rightarrow \alpha'$, functions by λ-notation. Support for datatype definitions like:

datatype α option = Some α | None

introducing the option type as an enumeration with the alternatives Some x and None, and the usual pattern matching notation give Isabelle/HOL a similar flavour like functional programming languages like F# or Haskell, except that the combined language comprises logical quantifiers and logical, extensional equality. Thus, it often allows a very natural way of specification. Isabelle/HOL provides also a large collection of theories like pairs, sets, lists, multisets, maps, orderings, and various arithmetic theories. Of particular importance for the models described here is the type of *partial functions* $\alpha \rightharpoonup \beta$, modelled as synonym to functions $\alpha \Rightarrow \beta$ option, which also provides the usual concept of domain dom$f$ and range ran$f$ on them.

HOL-TestGen is an extension to Isabelle/HOL designed to support model-based testing. It offers support for the typical phases of a model-based testing process: 1) writing the *test theory*, i. e. a collection of basic types and auxiliary functions formalising the problem domain, 2) writing the *test specification*, TS, specifying the concrete property to be tested, 3) the *test case generation* phase, i. e. an automated conversion of TS into a sequence of *test cases*, TC, (or: partitions) representing classes of possible input, 4) the *test data generation* phase, during which concrete members are constructed for the TC, and 5) the *test execution* phase when HOL-TestGen generates a *test script* driving the actual testing. Once a test theory is completed, documents can be generated that represent a formal test plan, including test theory, test specifications, configurations of the test data and test script generation commands.

The core of the test case generation procedure lies in case splittings up to a certain depth for each free or universally quantified (input) variable in the test specification; depth and form of the case split depend on the type of the variable. The resulting test cases, TC$_i$, have the form $C_1\ x \wedge \cdots \wedge C_n\ x \rightarrow P(\mathsf{PUT}\ x)$, where PUT is a place-holder for the program under test, x is the input vector and P is the oracle or postcondition telling that the output of PUT complies with the test specification. Test data generation from test cases boils down to a constraint resolution process finding an x satisfying the constraints C_i. The reader interested in more details is referred to [11, 12].

2.3 Monads for Sequence Testing

Modelling and reasoning over computations requires mechanisms to deal with states and state transitions within the logic. HOL, however, as a purely functional specification formalism has no such built-in concepts. Using *monads*—a concept made popular for purely functional programming languages by Peyton Jones and Wadler [28]—is one way to overcome this apparent limitation. 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-Milner type system, it is not possible to represent monads *as such* in HOL, only concrete instances. We define such an instance for our purpose, the state-exception monad, which models precisely partial state transition functions of type

types $(\mathsf{o}, \sigma)\ \mathsf{MON}_{\mathrm{SE}} = \sigma \rightharpoonup (\mathsf{o} \times \sigma)$

Using monads, we can view our programs under test, PUT, as *i/o stepping functions* of type $\iota \Rightarrow (\mathsf{o},\sigma)\mathsf{MON}_{\mathrm{SE}}$, where each stepping function may either fail for a given state σ and input ι, or produce an output o and a successor state.

The usual concepts of *bind* (representing sequential composition with value passing) and *unit* (representing the embedding of a value into a computation) are defined for the case of the state-exception monad as follows:

definition $\mathsf{bind}_{\mathrm{SE}} :: (\mathsf{o},\sigma)\mathsf{MON}_{\mathrm{SE}} \Rightarrow (\mathsf{o} \Rightarrow (\mathsf{o}',\sigma)\mathsf{MON}_{\mathrm{SE}}) \Rightarrow (\mathsf{o}',\sigma)\mathsf{MON}_{\mathrm{SE}}$
where $\mathsf{bind}_{\mathrm{SE}}\ \mathsf{f}\ \mathsf{g} \equiv \lambda\sigma.$ **case** f σ **of** None $\Rightarrow$ None | Some(out, σ') $\Rightarrow$ g out σ'
definition $\mathsf{unit}_{\mathrm{SE}} :: \mathsf{o} \Rightarrow (\mathsf{o}, \sigma)\mathsf{MON}_{\mathrm{SE}}$
where $\mathsf{unit}_{\mathrm{SE}}\ \mathsf{e} \equiv \lambda\sigma.\ \mathsf{Some}(\mathsf{e},\sigma)$

We write x←f; g for $\mathsf{bind}_{\mathrm{SE}}$ f(λ x. g) and return for $\mathsf{unit}_{\mathrm{SE}}$. On this basis, the concept of a *valid test sequence* can be specified:

$\sigma \models o_1 \leftarrow \mathsf{PUT}\ i_1;\ \ldots;\ o_n \leftarrow \mathsf{PUT}\ i_n;\ \mathsf{return}\ (\mathsf{P}\ o_1\ \ldots\ o_n)$

where $\sigma \models \mathsf{m}$ is defined as $(\mathsf{m}\ \sigma \neq \mathsf{None} \wedge \mathsf{fst}\,(\mathsf{the}\ (\mathsf{m}\ \sigma)))$ and where the (Some x) = x. For iterations of i/o stepping functions, we also use an mbind operator, which takes a list of inputs $\iota\mathsf{s} = [i_1,\ldots,i_n]$, feeds it subsequently into PUTand stops when an error occurs. Using mbind, valid test sequences can be reformulated by:

$\sigma \models \mathsf{os} \leftarrow \mathsf{mbind}\ \mathsf{PUT}\ \ \iota\mathsf{s};\mathsf{return}\,(\mathsf{P}\ \mathsf{os})$

which is the standard way to represent sequence test specifications in HOL-TestGen.

3. MODELLING POLICIES IN HOL: UPF

In this section, we present the Unified Policy Framework (UPF), a generic framework for the modular modelling of testable security policies.

3.1 Foundation: Policies as Functions

We model the concept of a policy by partial policy decision functions. *Partial* functions are used since we describe elementary policies by partial system behaviour, which are glued together by operators such as function override and functional composition. In more detail, the partial policy decision functions are based on input data α (e. g., arguments, system state, security context) to output data β:

types $\alpha \mapsto \beta = \alpha \rightharpoonup \beta$ decision

where the enumeration type decision is defined as:

datatype α decision = allow α | deny α

This definition gives rise to a clear separation of the set of decisions into the allowance-set or A $\equiv$ {x | $\exists$ y. x = allow y} and the analogously defined deny-set D.

We introduce a number of common operations over policies p: these include *update*

$\mathsf{p}(\mathsf{x} \mapsto \mathsf{t}) \equiv \lambda \mathsf{y}.\ \ \mathsf{if}\ \ \mathsf{y} = \mathsf{x}\ \mathsf{then}\ \mathsf{Some}\ \mathsf{t}\ \mathsf{else}\ \ \mathsf{p}\ \mathsf{y}$

and its variants

$\mathsf{p}(\mathsf{x} {+}{\mapsto} \mathsf{t}) \equiv \mathsf{p}(\mathsf{x} \mapsto \mathsf{allow}\ \mathsf{t})$ and $\mathsf{p}(\mathsf{x} {-}{\mapsto} \mathsf{t}) \equiv \mathsf{p}(\mathsf{x} \mapsto \mathsf{deny}\ \mathsf{t})$

Elementary policies are

$\emptyset$	$\equiv \lambda$ x. None	
$\forall A_f$	$\equiv \lambda$x. Some (allow f x)	(*AllowAll*)
$\forall D_f$	$\equiv \lambda$x. Some (deny f x)	(*DenyAll*)

The operation *override* $\_\bigoplus\_ :: [\alpha \rightharpoonup \beta, \alpha \rightharpoonup \beta] \Rightarrow \alpha \rightharpoonup \beta$ allows elementary policies (*rules*) to be combined to more complex ones in a first-fit manner, e. g.:

$$p_1 \bigoplus \cdots \bigoplus p_n \bigoplus \forall D_f$$

where the last rule serves as "catch-all" (for a given function f producing the default return value, if any).

There are the notions of a *domain* dom p :: $[\alpha \rightharpoonup \beta] \Rightarrow \alpha$ set and a *range* ran p :: $[\alpha \rightharpoonup \beta] \Rightarrow \beta$ set. Inspired by the Z notation [31], we introduce the concept of *domain restriction* $\mathsf{S} \lhd \mathsf{p} \equiv \lambda\,\mathsf{x}.$ if x $\in$ S then p x else None and *range restriction* $\mathsf{p} \rhd \mathsf{S}$, defined analogously.

As an example, consider the Permission Assignment (PA) of Core RBAC [1]:

```
types PA_CoreRBAC = (ROLES ×PRMS) ↦unit,
```

where the type unit (not to be confused with the unit operator for monads) consists of only one element written (). It is used for policies having no output.

As another example, consider firewall policies as introduced in [13]:

```
types FP = packet ↦packet,
```

which also covers network address translations, i. e. the policy may allow a certain packet only after modification of its source or destination address.

3.2 Combining Policies

In the "policy-as-function"-view, policies are easy to combine using (higher-order) combinators. A number of generic combinators is provided in the UPF for common situations. There is a wide range of different semantic flavours that can be used in combination (similar to the policy combining algorithms of the eXtensible Access Control Markup Language (XACML) [27]). Consider the following semantics for combining two policies. Each of these can be desirable in different applications:

- First defined rule applies (this corresponds to the override $\_\bigoplus\_$ discussed previously)
- If allowed by any policy, return allow (called $\_\bigotimes_{\vee A}\_$)
- If denied by any policy, the decision is deny (called $\_\bigotimes_{\vee D}\_$)
- Only allow if allowed by both policies (called $\_\bigotimes_{\wedge A}\_$)
- Only deny if denied by both policies (called $\_\bigotimes_{\wedge D}\_$)

The latter four policies are inspired by parallel composition of automata: each policy makes an (independent) step, and the result is a step relation on the Cartesian product of states. As an example, consider *parallel-or-deny*:

```
definition _⨂∨D_ :: (α↦β) ⇒(γ↦δ) ⇒(α×γ↦β×δ)
 where   p1 ⨂∨D p2 = (λ(x,y). (case p1 x of
         Some (allow d1) ⇒ (case p2 y of
                           Some (allow d2) ⇒ Some (allow (d1,d2))
                         | Some (deny d2) ⇒Some (deny (d1, d2))
                         | None ⇒None)
       | Some (deny d1) ⇒  (case p2 y of
                           Some (allow d2) ⇒ Some (deny (d1, d2))
                         | Some (deny d2) ⇒Some (deny (d1, d2))
                         | None ⇒None)
       |None ⇒None))
```

The other cases proceed analogously. Since policies may have output in general, another fundamental combination concept is sequential composition. Similar to the parallel composition, four essential cases are to be distinguished and lead to the composition operators $\_\circ_{\vee D}\_$, $\_\circ_{\vee A}\_$, $\_\circ_{\wedge D}\_$, and $\_\circ_{\wedge A}\_$, which all have the type: $(\beta{\mapsto}\gamma){\Rightarrow}(\alpha{\mapsto}\beta){\Rightarrow}(\alpha{\mapsto}\gamma)$.

It may be necessary to adapt the input type or output 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 $\alpha \Rightarrow \beta$. The same effect can be achieved for the range with the ∇-operator defined by:

```
(f,g) ∇ p ≡λ x. case p x of
                    Some(allow x) ⇒ Some(allow (f x))
                  | Some(deny x) ⇒Some(deny (g x))
                  | None ⇒ None
```

It turns out that the two families of product and sequence operators are essentially enough to define a core language, which enjoys a number of properties, which we proved formally in Isabelle: For all sum, product, and sequential operators $\emptyset$ is neutral (i. e. $\mathsf{p} \bigotimes_{\vee D} \emptyset = \emptyset \bigotimes_{\vee D} \mathsf{p} = \emptyset$), all these operators are associative, enjoy some form of commutativity or pseudo-commutativity (i. e. $\mathsf{p} \bigoplus \mathsf{q} = \mathsf{q} \bigoplus \mathsf{p}$ if dom p and dom q are disjoint, or $\mathsf{p} \bigotimes_{\vee D} \mathsf{q} = (((\lambda(\mathsf{x},\mathsf{y}).\ (\mathsf{y},\mathsf{x}))\ \mathsf{o}_f\ (\mathsf{q} \bigotimes_{\vee D} \mathsf{p})))\ \mathsf{o}\ (\lambda(\mathsf{a},\mathsf{b}).(\mathsf{b},\mathsf{a}))$ where $\mathsf{g}\ \mathsf{o}_f\ \mathsf{p}$ is an abbreviation for $(\mathsf{g},\mathsf{g})\ \nabla\ \mathsf{p})$, and various forms of distributivity. All these algebraic laws established as derived rules give rise to a tool to normalise policies in order to simplify the task of an automated equivalence proof or test-case generation. For the special case of firewall policies, this technique has been successfully applied in [13] and has led to a breakthrough in the efficiency of the overall procedure.

3.3 Linking Transition Policies to Execution Sequences

A particular instance of the policy concept is the second-order *transition policy* of the form:

$$\iota{\times}\sigma{\mapsto}\mathsf{o}{\times}\sigma$$

where σ refers to some system state. Such transition policies are isomorphic to $\iota{\Rightarrow}(\mathsf{o}\ \mathsf{decision}\ ,\sigma)\ \mathsf{MON}_{\mathrm{SE}}$; thus, *i/o stepping functions* and *transition policies* are closely linked concepts and there are two conversion functions linking these two. Here, we will only present:

```
definition  policy2MON::(ι×σ)↦(o×σ) ⇒ι⇒(o decision,σ)MON_SE
where policy2MON p = (λ ισ. case p (ι,σ) of
          Some (allow (o, σ')) ⇒  (Some (allow o, σ'))
        | Some (deny (o, σ')) ⇒   (Some (deny o, σ'))
        | None ⇒  None)
```

It is easy to check that policy2MON is a bijection to state exception monads of the form: $(\mathsf{o}\ \mathsf{decision}\ ,\sigma)\mathsf{MON}_{\mathrm{SE}}$. As mentioned earlier, this link is import for the sequence testing approach of HOL-TestGen (see Section 5).

4. MODELLING NPFIT POLICIES

In this section, we show the instantiation of the UPF with a model of the Information Governance principles of the NPfIT. These principles are a typical example of a complex and realistic access-controlled system consisting of a range of different policy concepts. Thus, the instantiation allows us to present a typical usage of the UPF and proves its applicability in real-world scenarios.

While the individual policy parts will be quite different in other systems, the employed modelling strategy is typical and shows how the UPF can often be used. Abstractly speaking, the 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 ($\iota \times \sigma \mapsto$ 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, one for the exceptional behaviour.
- Use the combinators of the UPF to combine these parts (according to the desired test scenario, different combinations might be desired).
- Transform the combined policy into a state-exception monad to enable use of HOL-TestGen's sequence testing framework.

In the following, we only briefly outline how the individual parts of the systems can be modelled, and focus on their combination.

4.1 NPfIT Concepts and Operations

First, we describe how we model those main concepts of the NPfIT system which are relevant for the policy. The most important part is the set of Summary Care Records (SCRs), as the IG principles mainly govern access to them. An SCR is modelled containing some basic information including various status flags, plus references to demographic information (held by the Personal Demographics Service (PDS)), and to the proper care record content. Only last of these is governed by the concepts of LR, PC, and can be sealed. PC information is held in the main SCR, while seals are part of the content entries.

```
record entry =  entry_id         ::  entry_id
                entry_type       ::  entry_type
                seal             ::  seal
                provider         ::  user
                entry_content    ::  content
record SCR = patient_id  ::  patient
             flag        ::  consent_flag
             GP          ::  user
                          ⋮
             PDS         ::  entry_id  ⇀ PDS_entry
             content     ::  entry_id  ⇀ entry
```

We view the *Spine* as a partial function from patients to SCRs. We do not use it in the same meaning as in the NPfIT terminology.

```
types Spine = patient ⇀ SCR
```

Whenever a user wants to access a system function, he needs to present a User Role Profile (URP). While a user may have several URPs, only one at a time is active during an application session. URPs are defined by a record:

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

The *user context* υ stores all attributed URPs. This is needed as a user may only access the system with a URP that belongs to him:

```
types υ = user ⇀ (urp set)
```

Next, we model 29 operations, which are generic abstractions of functional behaviours governed by the 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 (see 4.4). 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. Here we only show a selection of them:

```
datatype Operation = createSCR urp patient name address
                               dob consent_flag  user
                   | extendSCR urp patient entry
                   | readSCR urp patient
                   | deleteSCR urp  patient
                   | removeEntry urp  patient  entry_id
                   | editEntry  urp  patient  entry_id  entry
                   | readEntry  urp  patient  entry_id
```

In some cases we might be interested in the output of an operation. We model the possible outputs as enumeration, including one element with an arbitrary string and the possibility to concatenate several outputs.

```
datatype Output = OutEntry entry | OutSCR SCR | ...
        | OutMsg string |  Conc Output Output (infixl  $ 80)
```

As an example (see Section 2.1.5), 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)]
```

4.2 NPfIT RBAC

In this section, we formalise the RBAC part of the NPfIT policy. As described in 2.1.1, the main ingredients of RBAC are User Role Profiles (URPs), roles, Areas of Work (AoW), activities, and functions.

The policy consists 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 relatively static and the same for every application, while the last one is application-dependent. All of them are modelled as simple relations.

Depending on the test scenario, we need to 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 Information Governance (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.

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 (e. g. 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).

4.3 Patient Consent (PC)

The concept of Patient 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: If the wish of the patient is unknown. Currently, this is interpreted as ask.

The rules about patient consent have the following type:

```
types PCPolicy = (Operation ×Spine) ↦unit
```

As only a limited number of operations is governed by these rules, we specify the set PC_Relevant_Ops.

Next, we define the individual rules modelling the desired semantics of the consent flag. As an example the following one 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
           Some s ⇒ (case  flag  s  of dontask ⇒ Some (allow ())
                                  |        _ ⇒ None)
           | _ ⇒ None)
   else  None))
```

Here, SCROp (op,sp) returns Some SCR as specified by the input of the operation op and the Spine sp or None if it does not exist. The other rules are similar and the full PC policy is the override ($\bigoplus$) of all these rules, with the default AllowAll rule for the non-matching inputs.

4.4 Legitimate Relationship (LR)

There are ten different types of Legitimate Relationships (LR), which need to be distinguished:

```
datatype LR_Type = PatientReferral | SelfClaimed (  ...  )
```

The LRs also have a status, which can take any of the following values:

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

The workgroups are sets of User Role Profiles (URPs) and have a unique identifier. While eight of the LRs are between a patient and a workgroup, two of them are between a patient and a single user. This is modelled as follows:

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

An LR is a record containing the following elements:

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

The policy about Legitimate Relationships (LR) needs as context information all the existing LRs and the workgroup memberships. These are stored in a security context Σ:

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

An LR policy is of the following type:

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

The function hasLR returns True if the given user has an active LR of any type with a specific patient in given security context. A typical rule about the concept of LRs then looks as follows:

```
LRPolicy1 ((editEntry  u p  e_i  e),  Σ) = (if  hasLR u p Σ
               then Some (allow ())  else  Some (deny ()))
```

Other rules (mainly those about how LRs can be transferred) additionally need to take the concrete type and status of an LR into account. Again, all the LR rules can be combined straightforwardly.

As an example, user John in the example mentioned earlier will only be able to read Pablo's SCR, if he is in a workgroup which has an active LR to Pablo. Here, this is achieved by Bob adding him to workgroup 1 just before.

4.5 Sealed Envelopes (SE)

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 Sealed Envelopes (SE) use information from the care records and the workgroup memberships. Thus, their type is:

```
types 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
                     None   ⇒ None
                   | Some e ⇒ ( if  (userHasAccess u wgs e)
                                          then Some (allow ())
                                          else  Some (deny ())))
| x ⇒ None)
```

The other rules are similar. 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.

4.6 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 usually 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. The output is modelled similarly. In the following, we show an excerpt of the state transition for an allowed operation on the Spine.

```
fun ST_A_Spine :: (Operation ×Spine) ⇀ Spine
where ST_A_Spine ((extendSCR u p e), S) =
       (case S p of None ⇒Some S
                | Some x ⇒Some (S(p↦x(|content := (content x)
                  ((SOME y. y ∉(dom (content x)))↦e)|))))
```

The individual state transitions can be combined to a single big one using the UPF operators. The following, e. g., is a model of the system behaviour if there were no policy:

$$\begin{array}{l}\textbf{definition}\ \ \mathsf{ST\_Allow} :: \\ \mathsf{Operation} \times \mathsf{Spine} \times \Sigma \times \upsilon \rightharpoonup \mathsf{Output} \times \mathsf{Spine} \times \Sigma \times \upsilon \\ \textbf{where} \\ \mathsf{ST\_Allow} = ((\mathsf{OUTPUT\_A}\ \mathsf{p\_m}\ (\mathsf{ST\_A\_Spine}\ \mathsf{o\_st}\ \mathsf{ST\_A\_}\Sigma \\ \qquad\qquad\qquad\qquad\qquad\qquad \mathsf{o\_st}\ \ \mathsf{ST\_A\_}\upsilon)) \\ \qquad\qquad \mathsf{o}\ (\lambda\ (\mathsf{a,b,c,d}).\ \ ((\mathsf{a,b}),(\mathsf{a,b,c,d}))))\end{array}$$

where o_st and p_m are operators from the UPF for parallel combinations of state transitions or partial functions respectively. ST_Deny is defined similarly.

4.7 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 to desired real behaviour. Despite the individual parts being rather distinct from each other, their combination is quite easy using the operators of the UPF. First, all the policy parts can be combined:

$$\begin{array}{ll}\textbf{definition} & \mathsf{appPolicy}\ ::\ (\mathsf{Operation} \times \mathsf{Spine} \times \Sigma \times \upsilon) \mapsto \mathsf{unit} \\ \textbf{where} & \mathsf{appPolicy} = \mathsf{C}_1\ \mathsf{o}_f\ ((\mathsf{C}_1\ \mathsf{o}_f\ ((\mathsf{C}_1\ \mathsf{o}_f \\ & \qquad\qquad (\mathsf{PCPolicy} \\ & \qquad\qquad\quad \bigotimes_{\vee D}\ \mathsf{SEPolicy})\ \mathsf{o}\ \mathsf{C}_2\ ) \\ & \qquad\qquad\quad \bigotimes_{\vee D}\ \mathsf{LRPolicy})\ \mathsf{o}\ \mathsf{C}_3\ ) \\ & \qquad\qquad\quad \bigotimes_{\vee D}\ \mathsf{AppRBACPolicy})\ \mathsf{o}\ \mathsf{C}_4\end{array}$$

where AppRBACPolicy is the RBAC policy of some application being part of the NPfIT. The coercion functions $\mathsf{C}_1 = \lambda(\mathsf{a,b}).\ \mathsf{a}$, $\mathsf{C}_2 = \lambda(\mathsf{a,b,c}).\ ((\mathsf{a,b}),(\mathsf{a,b,c}))$, $\mathsf{C}_3 = \lambda(\mathsf{a,b,c}).\ ((\mathsf{a,b,c}),\mathsf{a,c})$ and $\mathsf{C}_4 = \lambda(\mathsf{a,b,c,d}).\ ((\mathsf{a,b,c}),(\mathsf{a,d}))$ serve to a mere 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 $\mathsf{C}_5 = \lambda\mathsf{a}.\ (\mathsf{a,a})$:

$$\begin{array}{l}\textbf{definition}\ \ \mathsf{app\_ST\_Policy}\ ::\quad (\mathsf{Operation} \times \mathsf{Spine} \times \Sigma \times \upsilon) \\ \qquad\qquad\qquad\qquad\qquad\quad \mapsto (\mathsf{Output} \times \mathsf{Spine} \times \Sigma \times \upsilon) \\ \textbf{where} \\ \mathsf{app\_ST\_Policy}\ \ = \mathsf{C}_1\ \mathsf{o}_f\ ((((\mathsf{appPolicy} \triangleright \mathsf{A}) \\ \qquad\qquad\qquad\qquad\qquad\qquad \bigotimes_{\vee A}\ (\forall A \mathsf{x}.\ \mathsf{ST\_Allow}\ \mathsf{x}))\ \mathsf{o}\ \mathsf{C}_5) \\ \qquad\qquad\qquad \bigoplus (((\mathsf{appPolicy} \triangleright \mathsf{D}) \\ \qquad\qquad\qquad\qquad\qquad\qquad \bigotimes_{\vee A}\ (\forall D \mathsf{x}.\ \mathsf{ST\_Deny}\ \mathsf{x})))\ \mathsf{o}\ \mathsf{C}_5)\end{array}$$

And, finally, transformed into a state transition monad:

$$\begin{array}{ll}\textbf{definition} & \mathsf{appMon} \\ \textbf{where} & \mathsf{appMon} = \mathsf{policy2MON}\ \mathsf{app\_ST\_Policy}\end{array}$$

Possible usages of such a monad are described in the next section.

5. SECURITY TESTING OF THE NPFIT

In this section, we discuss several test purposes and test scenarios resulting in different test specifications and briefly describe how the generated test cases can be used for testing Web Service-based applications.

5.1 Test Specifications and Test Data

From an application perspective, we can distinguish two types of test specifications, i. e. properties that the system under test should fulfil: first, test specifications that ensure certain "quality criteria" of the modelled policy (e. g. is the policy always defined) and, second, test specifications that ensure that the applications conform to the policy and are compliant to legal regulations such as the Caldicott Report [32], or the NHS Confidentiality Code of Practice [16].

We start by generating test cases showing that our policy meets some *basic quality criteria*. For example, for every potential access X, the evaluation of the policy should give a well-defined result, i. e. allow or deny:

$$\neg(\mathsf{PUT}\ \mathsf{X} = \mathsf{None})$$

Applying HOL-TestGen to this test specification results, among others, in the following test data representing arbitrary attempts to access the system:

$$\mathsf{PUT}((\mathsf{readEntry}\ \ \mathsf{urp1\_alice}\ \ \mathsf{patient1}\ \ 2),\sigma 0) = \mathsf{Some}(\mathsf{deny}())$$

As an example for a scenario testing a critical situation, we might want to validate that the personal GP of a patient is always allowed to read his SCR.

$$\begin{array}{l}[\![\mathsf{UC}\ \mathsf{u} = \mathsf{Some}\ \mathsf{urps\_u};\ \mathsf{urp\_u} \in \mathsf{urps\_u};\ \mathsf{Sp}\ \mathsf{patient} = \mathsf{scr}; \\ \ \ \mathsf{gp}\ \mathsf{scr}\ = \mathsf{u}]\!] \Longrightarrow \\ \mathsf{Policy}\,((\mathsf{readSCR}\ \mathsf{urp\_u}\ \mathsf{patient}),\mathsf{Sp},\mathsf{SC},\mathsf{UC}) = \mathsf{Some}(\mathsf{allow}())\end{array}$$

A similar scenario, but this time exploring not only a single state transition but a sequence thereof, is that the consent status of a patient who is at one point declared as deceased should never be allowed to change, unless the patient is undeceased (i. e. his death status was an error) by an earlier operation.

A policy specified in a wide range of formal and informal documents, guidelines, etc. is prone to be underspecified or to contain ambiguities. In the case of the NPfIT this has already been observed before [6]. 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.

For example an early ambiguity detected by Becker [6] is whether users are able to seal data they are not allowed to read. We can test this property as follows:

$$\begin{array}{l}[\![\mathsf{Policy}\ \ ((\mathsf{readEntry}\ \ \mathsf{u}\ \ \mathsf{p}\ \ \mathsf{e}),\ \ \mathsf{spine},\ \ \mathsf{sc},\ \ \mathsf{uc})\ = \mathsf{Some}\ (\mathsf{deny}\ ())]\!] \\ \Longrightarrow \mathsf{PUT}\ (\ \mathsf{createSeal}\ \ \mathsf{u}\ \ \mathsf{p}\ \ \mathsf{e}\ \ \mathsf{s},\mathsf{spine},\mathsf{sc},\mathsf{uc}) = \mathsf{Some}\ (\mathsf{deny}\ ())\end{array}$$

Generating test cases that *ensure the compliance to standards and regulations* like the principles from the Caldicott Report [32] or stipulations such as a requirement that test results can only be accessed after breaking a seal usually require test specifications for sequence tests. The general format of such sequence tests is:

$$\begin{array}{l}\sigma 0 \models \mathsf{os} \leftarrow \mathsf{mbind}\ \mathsf{is}\ \mathsf{PolicyMonad};\ \mathsf{return}\ \ (\mathsf{os}{=}\mathsf{X}) \Longrightarrow \\ \sigma 0 \models \mathsf{os} \leftarrow \mathsf{mbind}\ \mathsf{is}\ \mathsf{PUT};\ \ \mathsf{return}\ \ (\mathsf{os}\ =\ \mathsf{X})\end{array}$$

meaning that if the formalised policy returns X beginning in some state $\sigma 0$, so should the program under test.

Often, such tests are too general and we need to limit the possible inputs; e. g. consider the following test specification

$$\begin{array}{l}[\![\mathsf{users}\ \ \mathsf{is}\ \subseteq \{\ \mathsf{urp1\_alice}\ ,\ \ \mathsf{urp2\_alice}\ ,\ \ \mathsf{urp\_john}\ ,\ \ \mathsf{urp\_bob}\}; \\ \sigma 0 \models \mathsf{os} \leftarrow \mathsf{mbind}\ \mathsf{is}\ \mathsf{appMon};\ \mathsf{return}\ (\mathsf{os}\ =\ \mathsf{X})]\!] \Longrightarrow \\ \sigma 0 \models \mathsf{os} \leftarrow \mathsf{mbind}\ \mathsf{is}\ \mathsf{PUT};\ \ \mathsf{return}\ \ (\mathsf{os}\ =\ \mathsf{X})\end{array}$$

which can be used to generate test data for the small example introduced in Section 2.1.5. Here, only Alice, Bob and John are allowed to perform an operation. This test specification produces test data like the following:

$$\begin{array}{l}\sigma 0 \models \mathsf{os} \leftarrow \mathsf{mbind}\ [(\mathsf{readSCR}\ \mathsf{urp\_john}\ \mathsf{pablo}), \\ \qquad\qquad\qquad\quad (\mathsf{addToWG}\ \mathsf{urp\_bob}\ 1\ \{\mathsf{urp\_john}\}), \\ \qquad\qquad\qquad\quad (\mathsf{readSCR}\ \mathsf{urp\_john}\ \mathsf{pablo})]\ \ \mathsf{PUT}; \\ \quad \mathsf{return}\ \ (\mathsf{os}\ =\ [(\mathsf{deny}\ \mathsf{OutNo}),(\mathsf{allow}\ \mathsf{OutSuccess}), \\ \qquad\qquad\qquad\qquad (\mathsf{allow}\ \ (\mathsf{OutSCR}\ \mathsf{SCR\_pablo}))])\end{array}$$

specifying the output that the PUT must produce when receiving the three operations in sequence (here, John is first denied access to Pablo's SCR, but is later allowed after Bob has added him to the workgroup *surgery*).

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 (e. g. access only possible using smart cards).

5.2 Testing Web Services

HOL-TESTGEN supports the generation of test-scripts (written in SML) that allow for the automated testing of real implementations (see [14]) for details). In its current form, HOL-TESTGEN only supports the automated testing of local implementations, i. e. distributed services-based systems are not supported. As modern distributed applications support, in large parts, WSDL compliant Web Service interfaces, we extended the test-script generation of HOL-TESTGEN to support the testing of WSDL-compliant Web services using the .net platform. In more detail, we

- ported the framework that executes the test-scrips to F#, a member of the ML family that is supported by the .net plattform.
- using the WSDL support of the .net plattform, we generated client libraries that allow to access the Web services under test using F#.
- we ported the test-scripts generated by HOL-TESTGEN to F#. While this was done manually in our current prototype, we see no difficulties in developing a test-script generator that directly generates F#.

A simplified excerpt of the test-script testing a Web-service in our small example (cf Section 2.1.5) looks as follows:

```
let _ = System.Console.Write("Test␣Case␣38:")
let pre_38  = []
let post_38 = valid
  ((fun a -> a = [deny OutNo
        allow OutSuccess
        allow (OutSCR SCR pablo])), Unity)
  (fun a -> mbind sendToWS
        [readSCR urp john pablo
         addToWG urp bob 1 {urp john}
         readSCR urp john pablo])
let res_38  = HolTestGen.TestHarness.check
                  retlet pre_38 post_38
```

This setup paves the way for automated unit and sequence tests of WSDL-compliant Web services with HOL-TESTGEN.

6. CONCLUSION AND RELATED WORK

6.1 Related Work

With Barker [4], our Unified Policy Framework (UPF) shares 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 [4], 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.

While there is a large body of literature adapting access control models to the specific needs of health care systems in general, e. g. [3, 5, 6, 19, 24], only a few (i. e. [5, 6, 19]) discuss the particular needs of the NHS in England.

On the modelling part, the closest related work is that of Becker [5, 6], presenting a formal model of the Information Governance (IG) policy using the authorisation policy language Cassandra [7]. While his model does cover some more details such as the management of credentials, it lacks, compared with our model, 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 NHS cannot be modelled directly in traditional access control frameworks such as RBAC [30] or Bell-LaPadula [8]. This is an instance of a more general lesson; that real-world applications tend to be loosely inspired by abstract frameworks rather than to implement them faithfully. Finally, Eyers et al. [19] implemented a NHS care record service that supports the runtime enforcement of the RBAC-based sub-policies.

As regards testing, the closest related works are those of Hu et al. [21], Martin and Xie [25] presenting a test case generation approach (based on change-impact analysis, respectively, mutation testing) for a subset of XACML [27] and that of Hu and Ahn [20] presenting a conformance testing approach for RBAC models using SAT solving techniques. Traon et al. [33] present a conformance testing approach that generates test cases for checking that an OrBAC [23] policy (an extension of RBAC also modelling the organisational contexts) complies with high-level compliance goals. Finally, there are several approaches, e. g. [22, 29], applying combinatorial testing to RBAC models. All approaches have in common that they consider only RBAC models or simple extensions thereof. In contrast, we used a uniform framework for security policies that is expressive enough to model sophisticated real-world policies such as the ones of NPfIT.

6.2 Model-based Testing In The Real World

The work presented in this paper has reinforced our conviction that MBT tools have the potential to satisfy a genuine practical need, in combination with complementary techniques such as penetration testing. 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, structured and subject to ongoing change;
- it applies to a range of application types from multiple providers.

Combined, these properties mean that the sizable investment of time and money in building the model could plausibly be justified in terms of mitigated risk, less problematic introduction of the software into service, and improved user experience.

However, building the model still requires specialised expertise, which remains a factor holding back the practical application of MBT. To improve this situation, more research in high-level graphical or textual languages and the use of reusable patterns and templates for the UPF is required. Integration of MBT tools into mainstream software development environments and testing suites would be a further way forward to enable their commercial use.

6.3 Conclusion and Future Work

We have presented a uniform framework for modelling security policies. This might be regarded as merely an interesting academic exercise in the art of abstraction, especially given the fact that underlying core concepts are logically equivalent, but presented remarkably different from—apparently simple—security textbook formalisations. However, we have successfully used the framework to model fully the large and complex information governance policy of a national health-care record system as described in the official documents (e. g. [17, 18]). Thus, we have shown the framework being able to accommodate relatively conventional RBAC mechanisms alongside less common ones such as Legitimate Relationships. These security concepts are modelled separately and combined into one global access control mechanism. Moreover, we have shown the practical relevance of our model by using it in our test generation system HOL-TestGen, translating informal security requirements into formal test specifications to be processed to test sequences for a distributed system consisting of applications accessing a central record storage system.

Besides applying our framework to other access control models, 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. Finally, this results in an improved test coverage.

References

[1] *American National Standard for Information Technology – Role Based Access Control.* INCITS 359-2004.

[2] R. Anderson. *Database State.* Joseph Rowntree Reform Trust Ltd, 2009. ISBN 9780954890247 (pbk.).

[3] C. A. Ardagna, S. D. C. di Vimercati, S. Foresti, T. W. Grandison, S. Jajodia, and P. Samarati. Access control for smarter healthcare using policy spaces. *Computers & Security*, 2010.

[4] S. Barker. The next 700 access control models or a unifying meta-model? In *SACMAT*, pages 187–196. ACM Press, 2009.

[5] M. Y. Becker. A formal security policy for an NHS electronic health record service. Technical Report UCAM-CL-TR-628, University of Cambridge, 2005.

[6] M. Y. Becker. Information governance in NHS's NPfIT: A case for policy specification. *International Journal of Medical Informatics*, 2007.

[7] M. Y. Becker. and P. Sewell. Cassandra: flexible trust management, applied to electronic health records. In *CSF*, pages 139–154. IEEE Computer Society, 2004.

[8] D. E. Bell and L. J. LaPadula. Secure computer systems: A mathematical model, volume II. In *Journal of Computer Security 4*, pages 229–263, 1996.

[9] S. Brennan. *The NHS IT project: the biggest computer programme in the world - ever!* Radcliffe Publishing, 2005.

[10] A. Browne. Lives ruined as NHS leaks patients' notes. The Observer, June 25th, 2000.

[11] A. D. Brucker and B. Wolff. Symbolic test case generation for primitive recursive functions. In J. Grabowski and B. Nielsen, editors, *FATES*, number 3395 in LNCS, pages 16–32. Springer, 2004.

[12] A. D. Brucker and B. Wolff. Test-sequence generation with HOL-TestGen – with an application to firewall testing. In B. Meyer and Y. Gurevich, editors, *TAP*, number 4454 in LNCS, pages 149–168. Springer, 2007.

[13] A. D. Brucker, L. Brügger, P. Kearney, and B. Wolff. Verified firewall policy transformations for test-case generation. In *ICST*, pages 345–354. IEEE Computer Society, 2010.

[14] A. D. Brucker, L. Brügger, M. P. Krieger, and B. Wolff. HOL-TestGen 1.5.0 user guide. Technical Report 670, ETH Zurich, 2010.

[15] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, 1940.

[16] Department of Health. Confidentiality. Code of Practice, 2003.

[17] Department of Health. The Care Record Guarantee. Our Guarantee for NHS Care Records in England, 2009.

[18] Department of Health. Information Governance (IG) Concepts, 2010. `http://www.connectingforhealth.nhs.uk/systemsandservices/infogov`.

[19] D. Eyers, J. Bacon, and K. Moody. OASIS role-based access control for electronic health records. In *IEE Software*, volume 153, pages 16–23. IEE, 2006.

[20] H. Hu and G.-J. Ahn. Enabling verification and conformance testing for access control model. In *SACMAT*, pages 195–204. ACM Press, 2008.

[21] V. Hu, E. Martin, J. Hwang, and T. Xie. Conformance checking of access control policies specified in XACML. In *COMPSAC*, volume 2, pages 275–280, 2007.

[22] V. Hu, D. Kuhn, and T. Xie. Property verification for generic access control models. In *EUC*, volume 2, pages 243–250, 2008.

[23] A. A. E. Kalam, S. Benferhat, A. Miège, R. E. Baida, F. Cuppens, C. Saurel, P. Balbiani, Y. Deswarte, and G. Trouessin. Organization based access control. In *POLICY*, pages 120–131. IEEE Computer Society, 2003.

[24] J. Longstaff, M. Lockyer, and M. Thick. A model of accountability, confidentiality and override for healthcare and other applications. In *Role-based access control*, pages 71–76. ACM Press, 2000.

[25] E. Martin and T. Xie. Automated test generation for access control policies via change-impact analysis. In *SESS*, pages 5–5, 2007.

[26] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL—A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[27] OASIS. extensible access control markup language (XACML), version 2.0, 2005.

[28] S. L. Peyton Jones and P. Wadler. Imperative functional programming. In *POPL*, pages 71–84. ACM Press, 1993.

[29] A. Pretschner, T. Mouelhi, and Y. Le Traon. Model-based tests for access control policies. In *ICST*, pages 338–347. IEEE Computer Society, 2008.

[30] R. Sandhu, V. Bhamidipati, and Q. Munawer. The ARBAC97 model for role-based administration of roles. *ACM TISSEC*, 2(1):105–135, 1999.

[31] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, Inc., 2nd edition, 1992.

[32] The Caldicott Committee. Report on the Review of Patient-Identifiable Information, 1997.

[33] Y. L. Traon, T. Mouelhi, and B. Baudry. Testing security policies: Going beyond functional testing. In *ISSRE*, pages 93–102, 2007.