Model-based Security Testing of a Health-Care System Architecture: A Case Study

MTV2 / Cacly NanoInnov, 12.11.2012

Achim Brucker (SAP), Lukas Brügger (ETH), Paul Kearney(BT) and Burkhart Wolff*

*Université Paris-Sud, Laboratoire de Recherche Informatique (LRI)

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 the NPfIT. NPfIT, "the National Program for IT" is a very large- scale development project aiming to modernise the IT infrastructure in the English health care system (NHS). Consisting of heterogeneous and distributed code, it is an ideal target for model-based testing techniques of a very large system exhibiting critical security features. We will 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 relationship. The model is given in higher-order logic (HOL) and processed together with suitable test-specifications in the HOL-TestGen system, that generates semi-automatically test sequences according to them.

Particular emphasis is put on a normalization technique of policies
--- demonstrated for reasons of IPR restriction at hand of firewall policies ---
which drastically improves the efficiency of the method.

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 the NPfIT. NPfIT, "the National Program for IT" is a very large- scale development project aiming to modernise the IT infrastructure in the English health care system (NHS). Consisting of heterogeneous and distributed code, it is an ideal target for model-based testing techniques of a very large system exhibiting critical security features. We will 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 relationship. The model is given in higher-order logic (HOL) and processed together with suitable test-specifications in the HOL-TestGen system, that generates semi-automatically test sequences according to them.

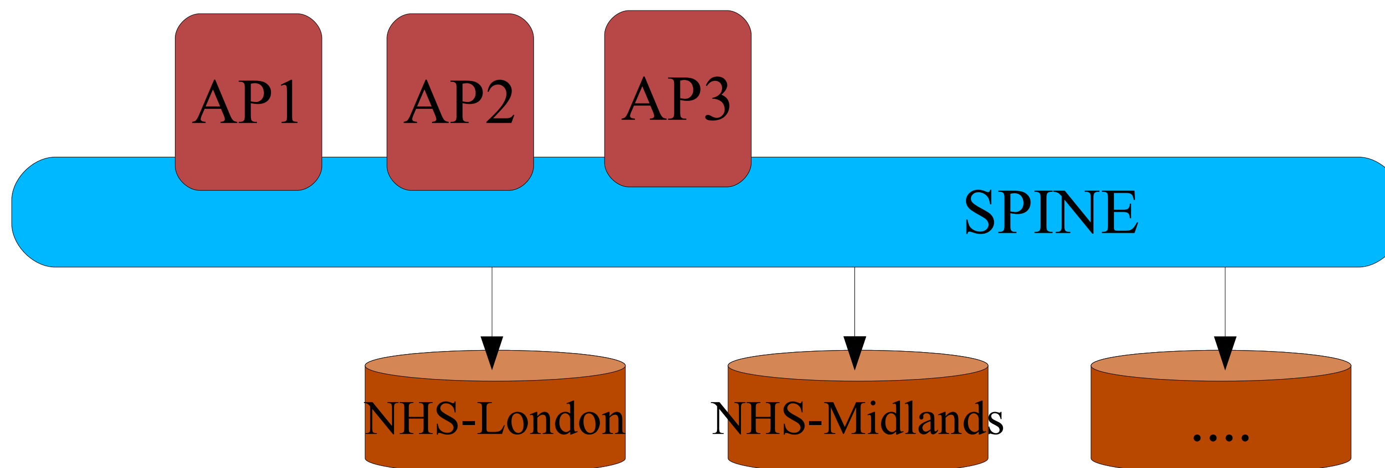
Particular emphasis is put on a normalization technique of policies
--- demonstrated for reasons of IPR restriction at hand of firewall policies ---
which drastically improves the efficiency of the method.

Overview

- NPfIT
- NPfIT formalized in UPF (formalized in HOL)
- System: HOL-TestGen
- Policy-Compression:
  an exercise in Test and Proof

National Program for IT (NPfIT)

- Large Case-Study together with British Telecom
- Test-Goal: NHS patient record access control mechanism
- Large Distributed, Heterogeneous System
- Legally required Access Control Policy (practically mostly enforced on the application level)



Case-Study: NPfIT

- Challenges:
  - 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 and RBAC are insufficiently expressive to capture complex policies such as Legitimate Relationships, Sealed Envelopes or Patient Consent Management.
  - 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 conventionally modeled in ABAC [6–8] and administrative RBAC [9, 10] models; A uniform modelling framework must be able to accommodate this.
  - The requirements are mandated by laws, official guidelines and ethical positions (e. g. [11, 12]) that are prone to change.

Case-Study: NPfIT

- Different "Information Gouvernance Principles" (= Policies):
  - Role-Based Access Control (RBAC): NPfIT uses administrative RBAC [9] to control who can access what system functionality. Each user is assigned one or more User Role Profile (URP). Each URP permits the user to perform several Activities.
  - Legitimate Relationship (LR): 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.
  - Patient Consent (PC): Patients can opt out in having a Summary Care Record (SCR) at all, or to control uploads of data into the SCR. This requires additional mechanisms to manage consent.
  - Sealed Envelope (SE): The sealing concept is used to hide parts of an SCR from users. Kinds of seals: seal, seal and lock, clinician seal.

Modeling Framework: Unified Policy Framework (UPF)

- UPF (A Theory in HOL / for HOL-TestGen)
  - A Policy: A Decision Function
    (Modeling a "Policy Enforcement Point" in a System)

datatype α decision = allow α | deny α

types (α,β) policy = α ⇀ β decision (* = α ⇒ β option *)

notation α ↦ β = (α,β) policy

Modeling Framework: Unified Policy Framework (UPF)

- UPF (A Theory in HOL / for HOL-TestGen)
  - Policy Constructors

definition $\varnothing \equiv \lambda\, y.\ \mathrm{None}$ (* $\varnothing :: \alpha \mapsto \beta$ *)

definition $p(x{+}{\mapsto}t) \equiv\ p(x \mapsto \mathrm{Some}(\mathrm{allow}\ t))$ (* $p :: \alpha \mapsto \beta$ *)
$p(x{-}{\mapsto}t) \equiv\ p(x \mapsto \mathrm{Some}(\mathrm{deny}\ t))$ (* where $p(x \mapsto t) \equiv$
$\lambda\, y.\ \text{if } y = x \text{ then } A \text{ else } p\ y$ *)

definition (*AllowAll :: "($\alpha \rightharpoonup \beta$) $\Rightarrow$ ($\alpha \mapsto \beta$)" *)

$\forall_A\, x.\ pf(x) \equiv (\lambda\, x.\ \text{case } pf\ x \text{ of } \mathrm{Some}\ y \Rightarrow \mathrm{Some}(\mathrm{allow}(y))$
$\mid \mathrm{None} \Rightarrow \mathrm{None})$

(*DenyAll :: "($\alpha \rightharpoonup \beta$) $\Rightarrow$ ($\alpha \mapsto \beta$)"*)

$\forall_D\, x.\ pf(x) \equiv (\lambda\, x.\ \text{case } pf\ x \text{ of } \mathrm{Some}\ y \Rightarrow \mathrm{Some}(\mathrm{allow}(y))$
$\mid \mathrm{None} \Rightarrow \mathrm{None})$

Modeling Framework: Unified Policy Framework (UPF)

- UPF (A Theory in HOL / for HOL-TestGen)
  - Domain, Range and Restictions on Policies (Z-like)

definition $A \equiv \{x.\exists y.\ x = \text{allow}\ y\}$, $D \equiv \{x.\exists y.\ x = \text{deny}\ y\}$

definition $\text{dom} :: \alpha \rightharpoonup \beta \Rightarrow \alpha\ \text{set}$
where $\text{dom}\ f \equiv \{x.\ f\ x \neq \text{None}\}$

definition $\text{ran} :: \alpha \rightharpoonup \beta \Rightarrow \beta\ \text{set}$ ...

definition $_ \lhd _ :: \alpha\ \text{set} \Rightarrow \alpha \rightharpoonup \beta \Rightarrow \alpha \rightharpoonup \beta$
where $S \lhd p \equiv (\lambda\ x.\ \text{if}\ x \in S\ \text{then}\ p\ x\ \text{else none})$ (* domain restriction *)

definition $_ \rhd _ :: \alpha \rightharpoonup \beta \Rightarrow \alpha\ \text{set} \Rightarrow \alpha \rightharpoonup \beta$ ... (* range restriction *)

definition $_ \oplus _ :: \alpha \rightharpoonup \beta \Rightarrow \alpha \rightharpoonup \beta \Rightarrow \alpha \rightharpoonup \beta$ ...(* first fit override *)

Example: Firewalls

- Firewall Policies in UPF
  - Data:

    ip-address = $int \times int \times int \times int$
    ip-packet = ip-address $\times$ protocol $\times$ content $\times$ ip-address

  - Firewall - Policies:

    policy : ip-packet $\mapsto$ ip-packet

    … this covers also Network Adress Translations (NAT's)

Example: Firewalls

- Firewall Policies in UPF
  - Elementary Policies

definition me-ftp :: ip-packet ↦ ip-packet
where me-ftp ≡ ∅ ((192,22,14,76),ftp,d,(192,22,14,76)
+↦(192,22,14,76),ftp,d,(192,22,14,76))

Example: Firewalls

- Firewall Policies in UPF
  - Elementary Policies

    definition me-ftp :: ip-packet $\mapsto$ ip-packet
    where me-ftp $\equiv$ $\varnothing$ ((192,22,14,76),ftp,d,(192,22,14,76)
    $+\!\mapsto$(192,22,14,76),ftp,d,(192,22,14,76))

  - Combined Policies:

    definition me-none-else:: ip-packet $\mapsto$ ip-packet

    where me-none-else $\equiv$ me-ftp $\oplus$ $\forall_D$ x. x

Example: Firewalls

- Firewall Policies in UPF
  - Elementary Policies

    definition me-ftp :: ip-packet $\Rightarrow$ ip-packet
    where me-ftp $\equiv$ $\varnothing$ ((192,22,14,76),ftp,d,(192,22,14,76)
    $+\mapsto$(192,22,14,76),ftp,d,(192,22,14,76))

  - Combined Policies:

    definition me-none-else:: ip-packet $\Rightarrow$ ip-packet

    where me-none-else $\equiv$ me-ftp $\oplus$ $\forall_D$ x. x

Example: RBAC

- RBAC Policies in UPF
  - Domain: $UR = users \times role$
    $RP = role \times permission$
  - 2-Policies:
    $UserTab :: UR \mapsto unit,$
    $PermTab :: permission \Rightarrow role \mapsto unit$

Example: RBAC

- RBAC Policies in UPF
  - Domain: $UR = users \times role$
    $RP = role \times permission$
  - 2-Policies:

    $UserTab :: UR \mapsto unit,$
    $PermTab :: permission \Rightarrow role \mapsto unit$

```
datatype  users  = …
datatype  roles  = …
datatype  permissions  = …

definition rbac  … RBAC (perm) = UserTab  o∨D PermTab(perm)
```

Example: RBAC

- RBAC Policies in UPF
  - Domain: $UR = users \times role$

    $RP = role \times permission$
  - 2-Policies:

    UserTab :: UR ⇛ unit,
    PermTab:: permission ⇛ role ⇛ unit

datatype users = …
datatype roles = …
datatype permissions = …

definition rbac … RBAC (perm) = UserTab $o_{\vee D}$ PermTab(perm)

Example: RBAC

- RBAC Policies in UPF
  - Domain: $UR = users \times role$

    $RP = role \times permission$
  - 2-Policies:

    UserTab :: UR ⇀ unit,
    PermTab:: permission ⇀ role ⇀ unit

datatype users = …
datatype roles = …
datatype permissions = …

definition rbac … RBAC (perm) = UserTab $o_{\vee D}$ PermTab(perm)

where $o_{\vee D}$ is one of the 4 policy sequential compositions

More on UPF

- Transition Policies
  - Transition Policies: Policies involving state

    $\alpha \times \sigma \mapsto \beta \times \sigma$ (input α, output β)

  - Higher-order Policies (Policies transforming policies)

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

  - Thus, ARBAC policies (policies describing who and how (1-order) policies may be modified) can be modelled in UPF

More on UPF

- Parallel Composition of Policies:
  - Idea: Considering policies as "transitions" in an automaton and putting them "in parallel" similar to automata composition.
  - Essentially 4 possibilities:

```
definition prod_orA ::"['α↦'β, 'γ ↦'δ] ⇒ ('α×'γ ↦ 'βx'δ)" (_ )
where "p1 ⊗∨A 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(allow(d1,d2))
                        | None ⇒ None)
               | Some(deny d1)⇒(case p2 y of
                         Some(allow d2)  ⇒ Some(allow(d1,d2))
                        | Some(deny d2)   ⇒ Some(deny (d1,d2))
                        | None ⇒ None)
               | None ⇒ None))
```

Principal Use of UPF for NPfIT

- Parallel Composition of 4 Policies + Functional:

$$
\begin{aligned}
&(norm_beh, excep_beh)\ \nabla \\
&\quad (legitimate_relation\ \otimes_{\vee A} \\
&\quad patients_consent\ \otimes_{\vee A} \\
&\quad sealed_envelopes\ \otimes_{\vee A} \\
&\quad rbac)
\end{aligned}
$$

NPfIT in UPF

- Test - Specifications:
  - Embedding of Transition Policies in State-Exception Monads:

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

NPfIT in UPF

- Test - Specifications:
  - Embedding of Transition Policies in State-Exception Monads:

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

Modeling Framework: Unified Policy Framework (UPF)

- State-Exception Monads(f.Test-Sequences in HOL)
  - State-Exception Monads:

    type (o,σ)MON_{SE} = σ ⇀ (o, σ)

    definition bind :: (o,σ)MON_{SE} ⇒ (o ⇒ (o,σ)MON_{SE}) ⇒ (o,σ)MON_{SE} (“ _ ; _ ← _”)
    where …

    definition unit :: (o ⇒ bool) ⇒ (o,σ)MON_{SE}) (“return _ ”)
    where …

  - Computation Sequences, Valid Computation Sequences, Valid mbind-Sequences, Valid mbind-Sequences with pre-condition:

    PUT(i_1) ; o_1 ← PUT(i_2); … ; on ← PUT(i_n) ; result(post o_1 … o_n)

    σ_0 ⊨ PUT(i_1) ; o_1 ← PUT(i_2); … ; on ← PUT(i_n) ; result(post o_1 … o_n)

    σ_0 ⊨ o_S ← mbind i_S PUT ; result(post o_S)

    pre i_S ⇒ σ_0 ⊨ o_S ← mbind i_S PUT; result(post o_S)

NPfIT in UPF

- Example for NPfIT:

  (General Pattern, formalizing an informal requirement) :

$$\text{pre } i_S \quad \Longrightarrow \quad \sigma_0 \models o_S \leftarrow \text{mbind PUT } (i_S);\ \text{result(post } o_S)$$

NPfIT in UPF

- Example for NPfIT:

  (General Pattern, formalizing an informal requirement) :

$$\llbracket \text{users } i_S \subseteq \{\text{urp1_alice}, \text{urp2_alice}, \text{urp_john}, \text{urp_bob}\}; \; \sigma_0 \models os \leftarrow \text{mbind } i_S \text{ RBAC_Mon}; \text{return } (os = X) \rrbracket$$

$$\Longrightarrow \quad \sigma_0 \models os \leftarrow \text{mbind } i_S \text{ PUT}; \text{return } (os = X)$$

Firewall Policies in UPF

- The UPF is conceived to capture a variety of different types of policies.

  So why not: Firewall- Poilicies ?

$$
\begin{aligned}
&\textbf{definition}\\
&\quad \text{Policy} \equiv \text{DenyAll}\\
&\qquad \oplus \text{AllowPort intranet internet } 80\\
&\qquad \oplus \text{AllowPort intranet dmz } 993\\
&\qquad \oplus \text{AllowPort dmz intranet } 25\\
&\qquad \oplus \text{AllowPort intranet dmz } 25\\
&\qquad \oplus \text{AllowPort internet dmz } 80\\
&\qquad \oplus \text{AllowPort internet dmz } 25
\end{aligned}
$$

Normalized Firewall Policies in UPF

- Observation: It is possible to make domain-separations of policies ("subnets") and eliminate redandant parts of a policy.

definition

$$\begin{aligned} \text{normalize}\, p \equiv (&\text{removeAllDuplicates} \circ \text{insertDenies} \\ &\circ \text{separate} \circ (\text{sort}\ (\text{Nets_List}\ \ p)) \\ &\circ \text{removeShadowRules2} \\ &\circ \text{remdups} \circ \text{removeShadowRules3} \\ &\circ \text{removeShadowRules1} \circ \text{policy2list})\ p \end{aligned}$$

Normalized Firewall Policies in UPF

- In HOL-TestGen, we can PROVE the correctness of this complicated, nine-phase normalizer:

```
theorem C_eq_normalize:
  assumes a1: member DenyAll p
      and a2: allNetsDistinct p
shows    C(list2policy(normalize p)) = C p
```

Empirical Results : Normalized (Firewall) Policies

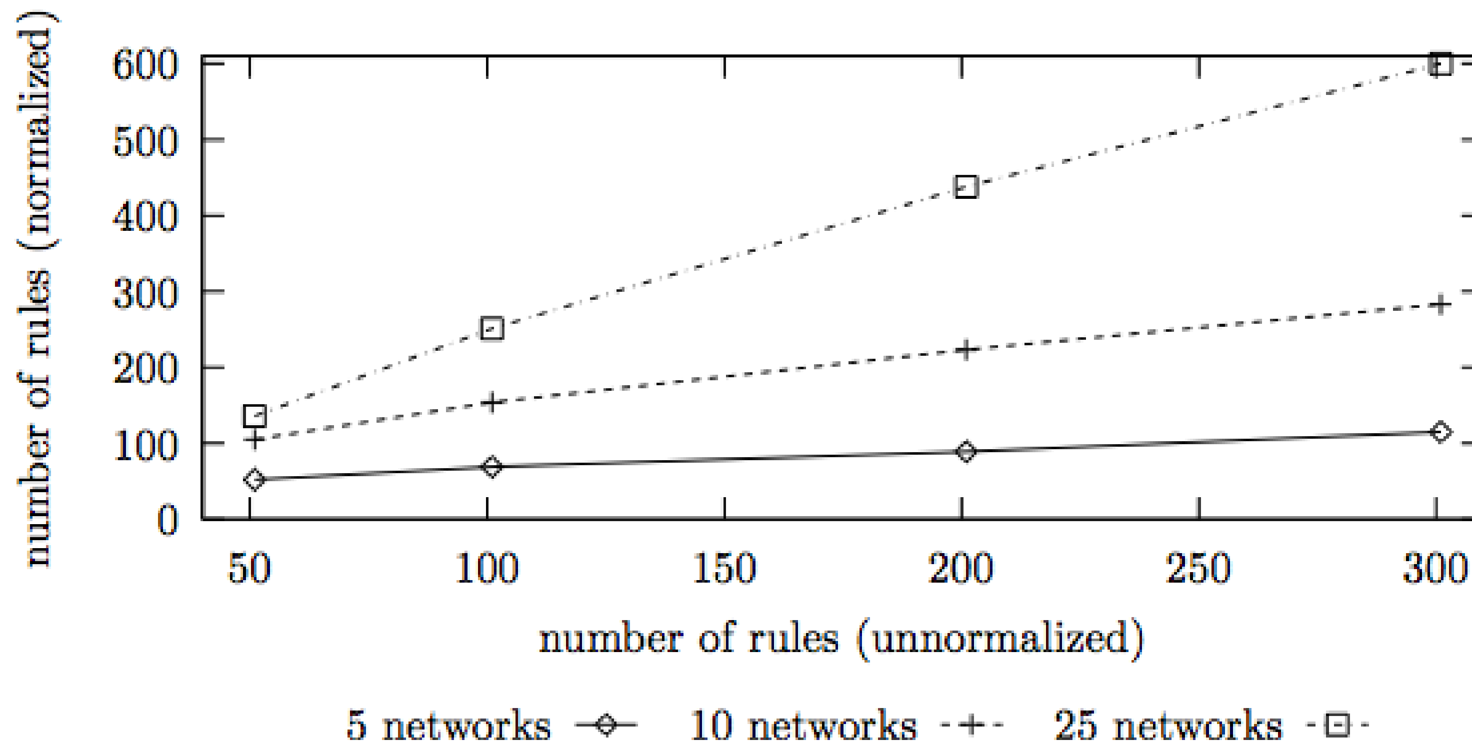
- Indeed, Normalization drastically simplifies the test-case generation:

		ETH 1	ETH 2	ETH 3	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
Not Normalized	Networks	4	6	3	2	3	4	4	4	3	5	5	6	6
	Rules	11	9	12	13	9	5	7	13	13	8	15	5	10
	TC Generation Time (sec)	>24h	22	26382	10	187	6	9	59364	1388	646	>24h	8	>24h
	Test Cases	—	100	1368	72	264	54	66	1544	470	358	—	54	—
Normalized	Rules	17	16	14	8	14	11	10	24	26	17	28	11	25
	Segments	6	5	4	2	4	5	3	7	4	6	9	5	9
	Normalization (sec)	0.5	0.3	0.6	0.5	0.4	0.2	0.3	1.1	0.8	0.3	1.4	0.2	0.8
	TC Generation Time (sec)	0.8	0.7	0.9	0.5	0.6	0.3	0.4	1.2	0.7	0.7	1.4	0.4	0.8
	Test Cases	22	22	20	12	20	12	14	34	22	22	38	14	32

Table II
TEST CASE GENERATION FOR FIREWALL POLICIES UTILIZING NORMALIZATION.

Empirical Results : Normalized (Firewall) Policies

- Indeed, Normalization drastically simplifies the test-case generation:



Our System: HOL-TestGen is ...

- ... based on HOL (Higher-order Logic):
  - "Functional Programming Language with Quantifiers"
  - plus definitional libraries on Sets, Lists, . . .
  - can be used meta-language for HoareCalculi, Z, CSP. . .
- ... implemented on top of Isabelle
  - an interactive prover implementing HOL
  - the test-engineer must decide over, abstraction level, split rules, breadth and depth of data structure exploration . . .
  - providing automated and interactive constraint-resolution techniques
  - interface: ProofGeneral
- ... by thy way, a verified test-tool

HOL-TestGen Workflow

- Modelisation
  - writing background theory of problem domain

HOL-TestGen Workflow

- Modelisation
  - writing background theory of problem domain
- Test-Case-Generation from Test-Specification
  - automated procedure gen_test_case ...
  - Test-Cases: partitions of I/O relation of the form

$$C_1(x) \Longrightarrow \ldots \; C_n(x) \Longrightarrow \text{post } x \; (\text{PUT } x)$$

HOL-TestGen Workflow

- Modelisation
  - writing background theory of problem domain
- Test-Case-Generation from Test-Specification
  - automated procedure gen_test_case ...
  - Test-Cases: partitions of I/O relation of the form

$$C_1(x) \Longrightarrow \ldots C_n(x) \Longrightarrow \text{post } x \text{ (PUT } x)$$

- Test-Data-Selection
  - constraint Solver gen_test_data
  - finds x satisfying $C_i(x)$

HOL-TestGen Workflow

- Modelisation
  - writing background theory of problem domain
- Test-Case-Generation from Test-Specification
  - automated procedure gen_test_case ...
  - Test-Cases: partitions of I/O relation of the form

$$C_1(x) \Longrightarrow \ldots C_n(x) \Longrightarrow \text{post } x \text{ (PUT } x)$$

- Test-Data-Selection
  - constraint solver gen_test_data
  - finds x satisfying $C_i(x)$
- Test-Driver Generation
  - automatically compiled, drives external program

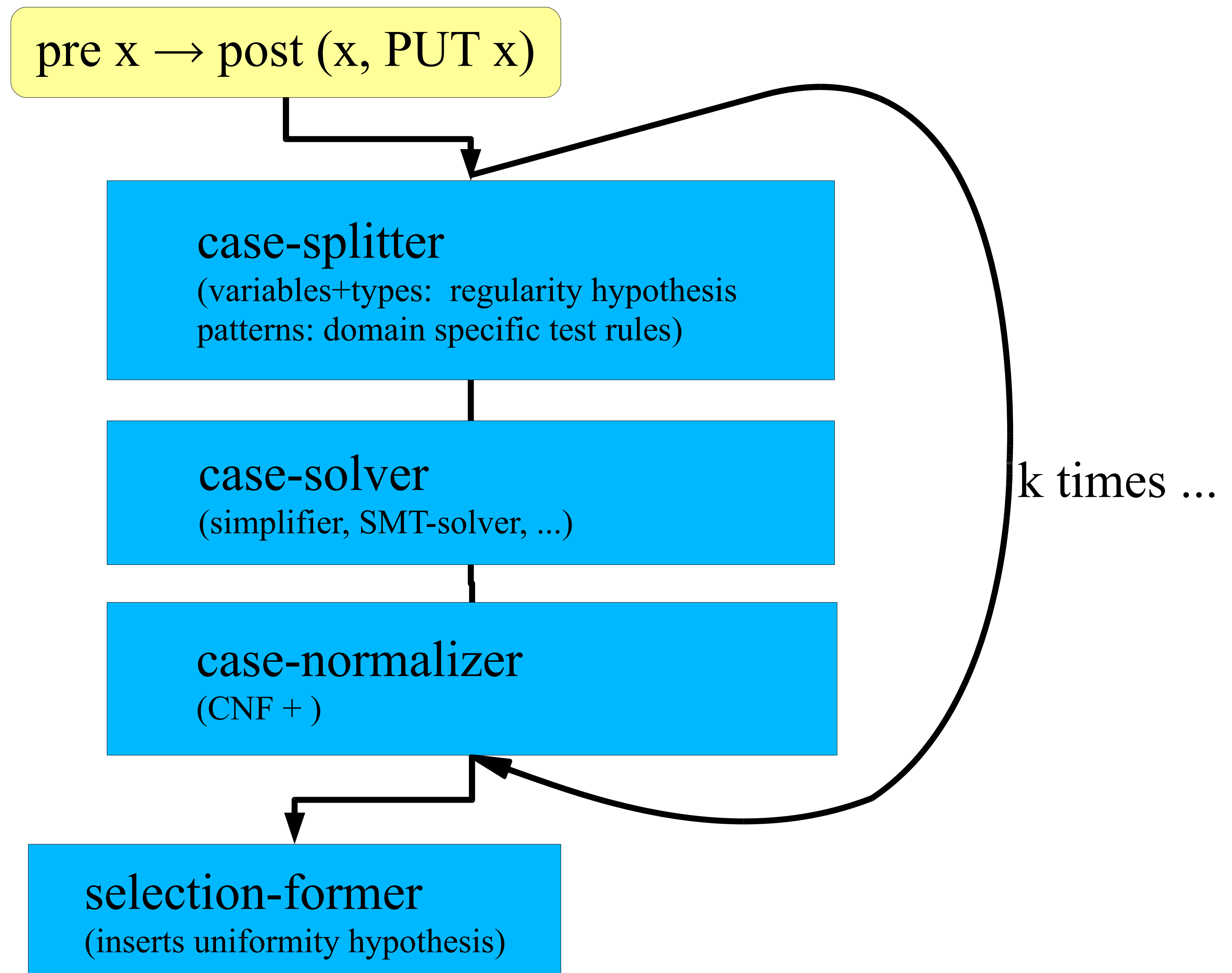
HOL-TestGen Workflow

- Modelisation
  - writing background theory of problem domain
- Test-Case-Generation from Test-Specification
  - automated procedure gen_test_case ...
  - Test-Cases: partitions of I/O relation of the form

$$C_1(x) \Longrightarrow \ldots C_n(x) \Longrightarrow \text{post } x \text{ (PUT } x)$$

- Test-Data-Selection
  - constraint solver gen_test_data
  - finds x satisfying $C_i(x)$
- Test-Driver Generation
  - automatically compiled, drives external program
- Test Execution, Test-Documentation

TestGen: Symbolic Computations



Demo

Scenario1.thy (modified)

Scenario1.thy (~/codebox/hol-testgen/add-ons/security/examples/firewall/CommonScenarios/Scenario1/)

```
exhibit the same behaviour as the modelled policy. *}

test_spec  "port_positive x ∧ accross_hosts x ∧ fix_values x  ⟶  FUT x = TestPolicy x"

txt{* The following command puts the test theorem into the desired form. *}
apply  (prepare_fw_spec)

txt{* Next, the policy is unfolded and possibly simplified *}
apply (simp add: policyLemmas)

txt{* Test case generation, takes about half a minute in this example: *}
apply (gen_test_cases "FUT")

txt{* Simplification of the generated test cases. This makes test data generation more efficient. *}
apply (simp_all add: policyLemmas)

txt{*St
store_t

text{*
externa
scenari
\texttt
\texttt
this to
indicat
solutio
```

206,1 (7019/849

Output

100% Tracing Auto update Update

```
proof (prove): step 6

goal (96 subgoals):
 1. FUT (1::int, (Host1, 1::int, udp), (Host2, 3389::int, tcp), data) = Some (allow ())
 2. FUT (1::int, (Host1, 1::int, udp), (Host2, 3389::int, udp), data) = Some (deny ())
 3. PO ((??X35X6 ≤ (4096::int) ∧ (1024::int) ≤ ??X35X6) ∧ ??X35X6 ≠ (3389::int))
 4. FUT (1::int, (Host1, 1::int, udp), (Host2, ??X35X6, tcp), data) = Some (allow ())
 5. THYP ((∃x≤4096::int.
              (1024::int) ≤ x ∧
              x ≠ (3389::int) ∧
              FUT (1::int, (Host1, 1::int, udp), (Host2, x, tcp), data) = Some (allow ())) ⟶
          (∀x≤4096::int.
              (1024::int) ≤ x ⟶
              x ≠ (3389::int) ⟶
```

Conclusion

- HOL-TestGen used for NPfIT was success wrt:
  - superior modeling techniques
  - substantial conservative libraries
  - standardized interfaces to tactic and automatic proof
  - code generation
  - a programming interface and genericity in design

  … offering lot of machinery not worth to reinvent.

Conclusion

- HOL-TestGen opens up new ways for combined efforts of test and proof:
  - abstraction leads to reusable modeling frameworks and generic normalization theorems
  - symbolic computation is an overhead, but leads to unforeseen opportunities for technical simplification and and ways to cope with state-explosion.