A Compositional Testing Framework Driven by Partial Specifications

TESTCOM/FATES 2007, Tallinn, Estonia

Ylies Falcone, Jean-Claude Fernandez, Laurent Mounier, Jean-Luc Richier

Verimag Laboratory

LIG Laboratory

Grenoble Universities

Friday, June 29

Testing a security policy ?

Inspired from **protocol conformance testing**:

- definition, generation and execution of **test cases**:
  - sequences of **interactions** between a testing tool and a SUT
  - deliver a **verdict** $\in$ {Pass, Fail, Inconc}
- test generation can be (partially) automated
- well-defined theory

But:

- needs a “complete” (functional) *specification* of the system under test,
- test interactions performed at a precise *interface level*

$\Rightarrow$ not the case when testing a security policy ...

Proposed approach ... (1)

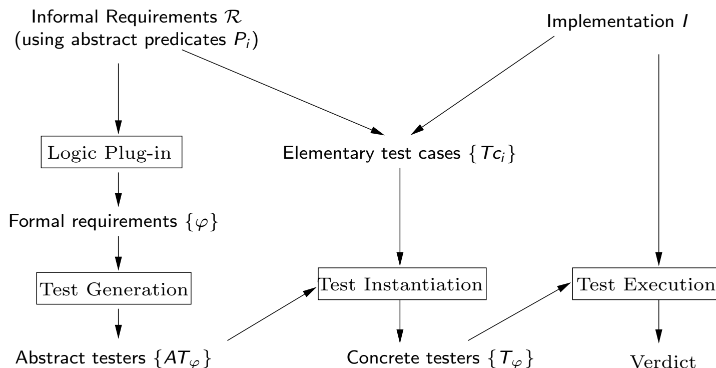
- security policy = set of requirements expressed by **logical formula**
- **property-oriented** test generation based on the structure of each formula φ

More precisely:

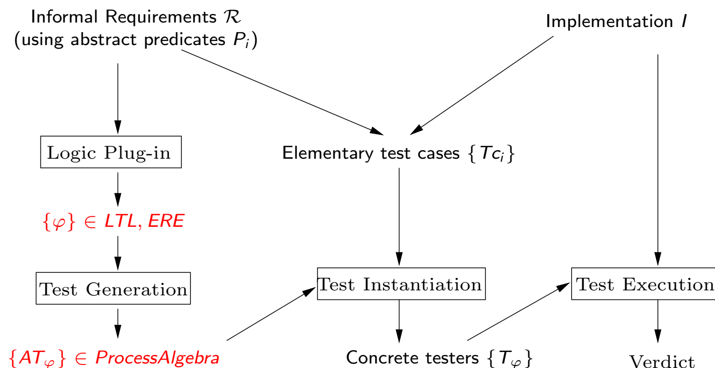
- elementary predicates of $\varphi \Rightarrow$ test pattern t_i, designed by administrators or security experts
- global test case t_φ obtained by connecting t_i with **test controllers** corresponding to logical operators of φ.

$\Rightarrow$ structural correspondance between formula and test cases

Proposed approach (2)



Proposed approach (2)



Outline

Test Process Algebra

Test cases: test processes manipulating data and interacting with the SUT

- Parallelise test executions
- Exception mechanism

Basic test cases

$$\begin{array}{lll} t & ::= & [b]\gamma \circ t \mid t + t \mid nil \mid recX\ t \mid X \\ b & ::= & \text{true} \mid \text{false} \mid b \vee b \mid b \wedge b \mid \neg b \mid expr_\tau = expr_\tau \\ \gamma & ::= & x_\tau := expr_\tau \mid !c(expr_\tau) \mid ?c(x_\tau) \end{array}$$

Test cases (test processes)

$$p ::= t \mid t \parallel_{cs} t \mid t \ltimes^{\mathcal{I}} t$$

Test Process Algebra

Test cases: test processes manipulating data and interacting with the SUT

- Parallelise test executions
- Exception mechanism

Basic test cases

$$
\begin{array}{lcl}
t & ::= & [b]\,\gamma \circ t \mid t + t \mid nil \mid recX\ \ t \mid X \\
b & ::= & \mathrm{true} \mid \mathrm{false} \mid b \vee b \mid b \wedge b \mid \neg b \mid expr_\tau = expr_\tau \\
\gamma & ::= & x_\tau := expr_\tau \mid !c(expr_\tau) \mid ?c(x_\tau)
\end{array}
$$

Test cases (test processes)

$$p ::= t \mid t \parallel_{cs} t \mid t \ltimes^{\mathcal{I}} t$$

Test verdicts

Test execution of t_φ delivers a **test verdict**:

- *Pass*: φ was not violated during the test execution
- *Fail*: φ was violated during the test execution
- *Inconc*: test execution did not allow to conclude about validity of φ

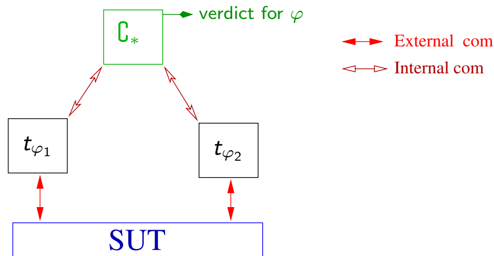
Test verdict computation:

- a "local" verdict is produced by each basic tester (basic predicates of φ)
- each controller combines its local verdicts and propagate the result "upward" (using communication channels)
- the final verdict is computed by the main controller

Principle : produce $t_\varphi = GenTest(\varphi)$

- basic tester t_{p_i} associated with each abstract predicate p_i
- each $\psi = F^n(\psi_1, \cdots, \psi_n) \longrightarrow$ test process t_ψ
  $\hookrightarrow t_\psi$: parallel composition of $t_{\psi_1}, \ldots, t_{\psi_n}$ and C_{F^n} called a test controller for operator F^n.

Example : $\varphi = \varphi_1 * \varphi_2$

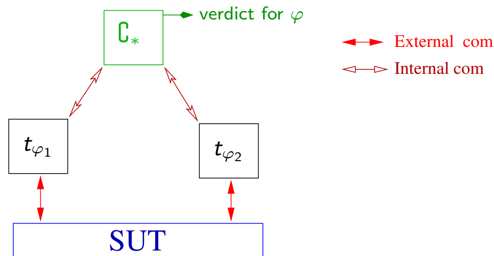


Basic tester:

1. test pattern associated to basic predicates of φ
2. depending on the control/observation level of the SUT

Controllers:

1. Test execution management (sub-tests scheduling) :
2. Verdict computation : "implement" logic operator semantics (one generic controler per operator of φ)



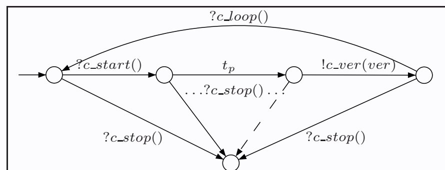
Test function generation

$$GenTest(\varphi) \stackrel{\text{def}}{=} GT(\varphi, cs) \parallel_{\{c_start, c_ver\}} (!c_start() \circ ?c_ver(x) \circ nil)$$

Basic case

$GT(p_i, cs) \stackrel{\text{def}}{=} Test(t_{p_i}, cs)$

$Test(t_p, \{c_start, c_stop, c_loop, c_ver\}) \stackrel{\text{def}}{=}$



Inductive case

$GT(F^n(\phi_1, \ldots, \phi_n), cs) \stackrel{\text{def}}{=}$

$(GT(\phi_1, cs_1) \parallel \cdots \parallel GT(\phi_n, cs_n)) \parallel_{cs'} \mathbb{C}_{F^n}(cs, cs_1, \ldots, cs_n)$

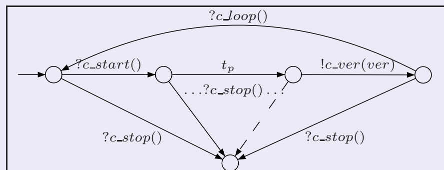
Test function generation

$$GenTest(\varphi) \stackrel{\mathbf{def}}{=} GT(\varphi, cs) \parallel_{\{c_start, c_ver\}} (!c_start() \circ ?c_ver(x) \circ nil)$$

Basic case

$$GT(p_i, cs) \stackrel{\mathbf{def}}{=} Test(t_{p_i}, cs)$$

$$Test(t_p, \{c_start, c_stop, c_loop, c_ver\}) \stackrel{\mathbf{def}}{=}$$



Inductive case

$$GT(F^n(\phi_1, \ldots, \phi_n), cs) \stackrel{\mathbf{def}}{=}$$

$$(GT(\phi_1, cs_1) \parallel \cdots \parallel GT(\phi_n, cs_n)) \parallel_{cs'} \mathbb{C}_{F^n}(cs, cs_1, \ldots, cs_n)$$

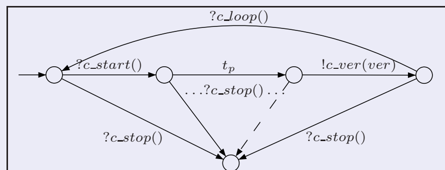
Test function generation

$$GenTest(\varphi) \stackrel{\textbf{def}}{=} GT(\varphi, cs) \parallel_{\{c_start, c_ver\}} (!c_start() \circ ?c_ver(x) \circ nil)$$

Basic case

$$GT(p_i, cs) \stackrel{\textbf{def}}{=} Test(t_{p_i}, cs)$$

$$Test(t_p, \{c_start, c_stop, c_loop, c_ver\}) \stackrel{\textbf{def}}{=}$$



Inductive case

$$GT(F^n(\phi_1, \ldots, \phi_n), cs) \stackrel{\textbf{def}}{=}$$
$$(GT(\phi_1, cs_1) \parallel \cdots \parallel GT(\phi_n, cs_n)) \parallel_{cs'} \mathtt{C}_{F^n}(cs, cs_1, \ldots, cs_n)$$

Outline

Instanciation on two "logics"

Testing framework instanciated:

- Action based LTL-X
- Extended Regular Expressions

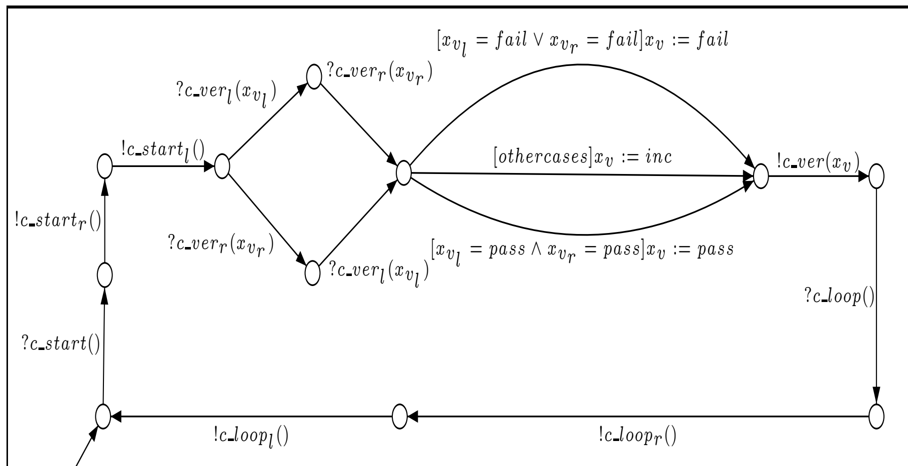
How instanciate this framework?

- Give *GenTest* function
- Prove its soundnesss

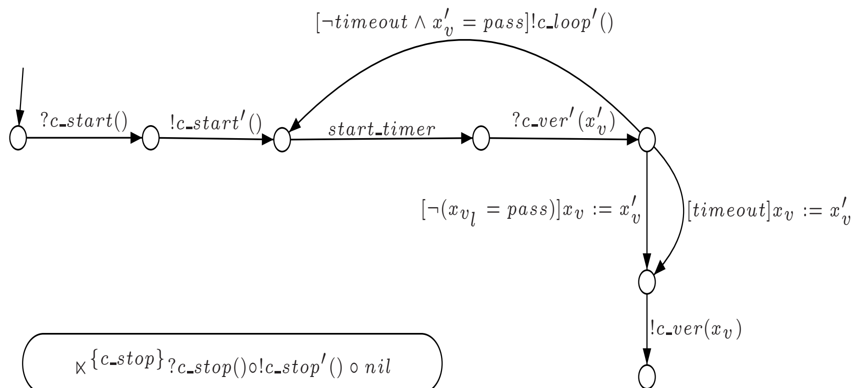
$\Longrightarrow$ *GenTest* is made explicit by giving controllers

- LTL-X : $\mathtt{C}_{\wedge}, \mathtt{C}_{\neg}, \mathtt{C}_{\mathcal{U}}$
- EREs : $\mathtt{C}_{\neg}, \mathtt{C}., \mathtt{C}choice, \mathtt{C}^{+}$

The $\mathbb{C}_\wedge$ controller (simplified)



The $\mathbb{C}_+$ controller (simplified)



Soundness result

Abstract test cases are always sound

Hypothesis needed: basic test cases are sound

Theorem

Let φ be a formula, and $t = GenTest(\varphi)$, σ a test execution sequence, the proposition is:

$$VExec(\sigma) = pass \Longrightarrow \sigma \textbf{ satisfies } \varphi$$
$$VExec(\sigma) = fail \Longrightarrow \sigma \textbf{ does not satisfy } \varphi$$

Outline

Test generation

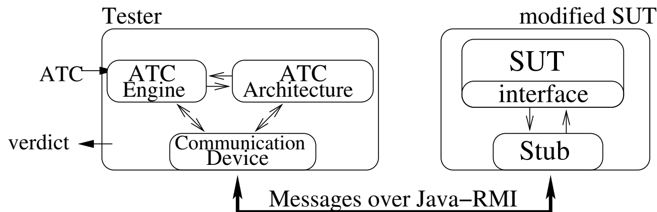
Inputs

- SUT interface : controllable and observable actions
- ATC library
  - writing test case
  - use SUT interface
- Requirement in a trace-based formalism

$\longrightarrow$ abstract test case corresponding to the requirement

Test Execution

Architecture overview:



"Black-box" approach : interface calls

Conclusion

Testing framework

- Produce and execute test cases
- High level requirement (trace-based formalism: LTL-X, EREs)
- syntax driven
- structured tests
- no need for a complete specification of the SUT
- ... but need some expertise to design the initial basic tester

Prototype tool, basic experiments

Current work

Implementation within a **test environment**

- test generation and test execution tools
- abstract test case library
- connection to existing security policy description languages (e.g., OrBac)
- more case studies . . .

Use MOP technology [Chen, Rosu 05]

- connection with monitoring technique
- integration abilities (abstract aspects)