Tools for Traceable Security Verification

Jan Jürjens*, Yijun Yu, Andreas Bauer

Computing Department
The Open University, GB

Computer Sciences Lab
The Australian National University

[*from 1 Oct 2008 also visiting at: Microsoft Research (Cambridge)]

J.Jurjens@open.ac.uk

Problem

How do I know a crypto-protocol implementation (as opposed to specification) is secure ?

Possible solution:

Verify specification, write code generator, verify code generator.

Problems:

- very challenging to verify code generator
- generated code satisfactory for given requirements (maintainability, performance, size, …) ?
- not applicable to existing implementations

Alternative Solution

Verify implementation against security requirements.

So far applied to self-written or restricted code.

Surprisingly few approaches so far:

- J. Jürjens, M. Yampolski (ASE´05, ASE'06, …): methodology + initial results for restricted C code
- J. Goubault-Larrecq, F. Parrennes (VMCAI´05): self-coded client-side of Needham-Schroeder in C
- K. Bhargavan, C. Fournet, A. Gordon (CSFW´06, …): self-coded implementations in F-sharp
- Reif, Schellhorn et al (forthcoming): self-constructed code

May reduce first problem (verify code generator). How about other two (requirements on code; legacy code)?

Towards Verifying Legacy Implementations

Goal: Verify pre-existing implementation. Options:

2) Generate models from code and verify these.
   - Advantages:
     -- Seems more automatic.
     -- Users in practice can work on familiar artifact (code), don´t need to otherwise change development process (!).
   - Challenges: Currently possible for restricted code or using significant annotations. Need to verify model generator.

2) Create models and code manually and verify code against models. Advantages:
   - Split heavy verification burden (Model-level analysis more efficient).
   - Get some verification result already in design phase (for non-legacy implementations) ➔ cheaper to fix.

Just an Exercise in Code Verification ?

State of the art in code verification in practice: execution exploration by *testing*. Limitations:

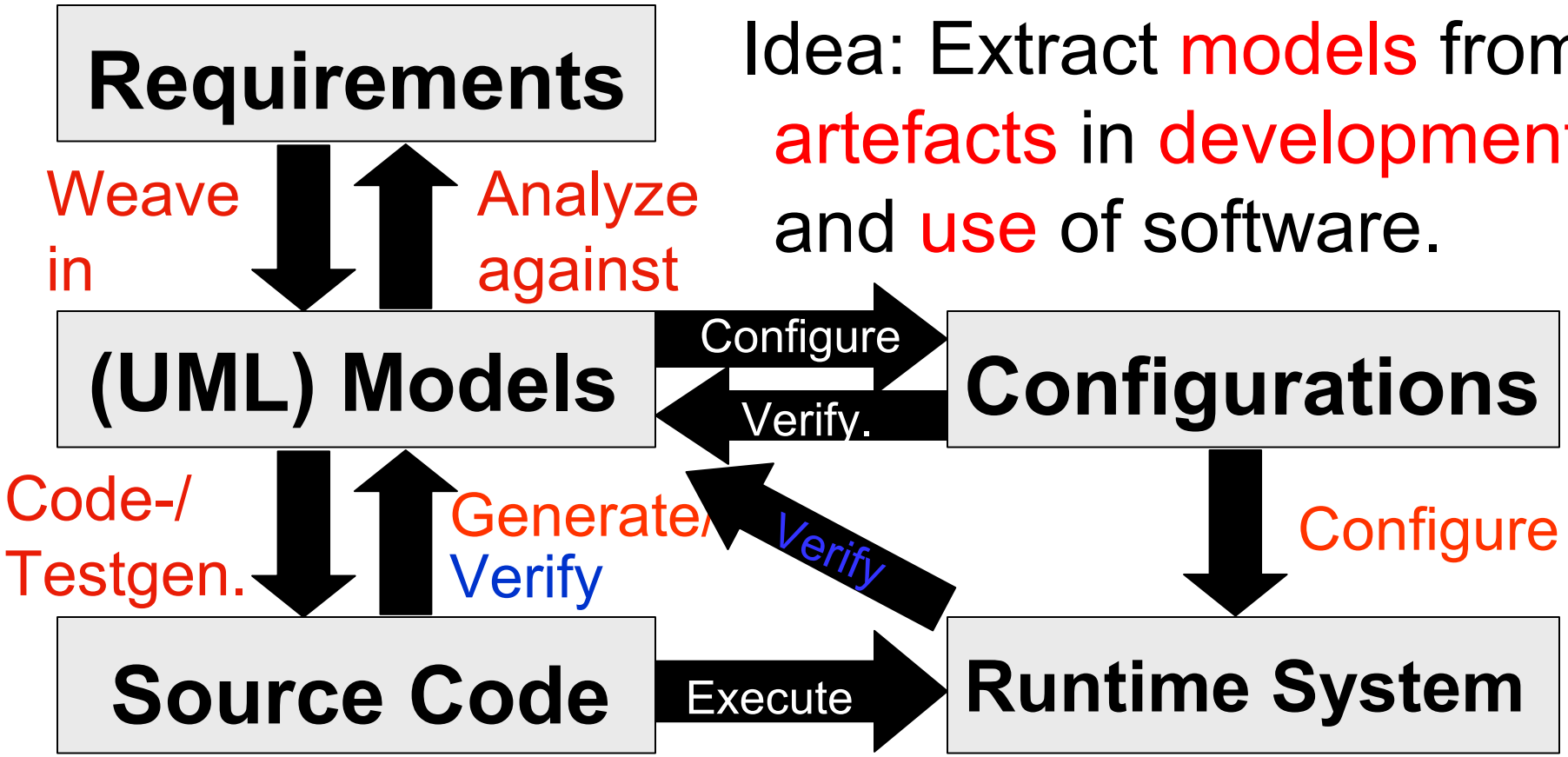
- For highly interactive systems usually only partial test coverage due to test-space explosion.
- Cryptography inherently un-testable since resilient to brute-force attack.

Interactive formal software verification (Isabelle et al): assumes specialist users.

Automated … (Bandera, Soot et al.): scalability wrt. code size / complexity; sophistication of properties (security).

➔ Develop specialized verification approach based on these.

Context: Model-based Security Engineering



Idea: Extract models from artefacts in development and use of software.

➔ Long-term goal: Tool-supported, theoretically sound, efficient automated security design & analysis.

Security Analysis in First-order Logic

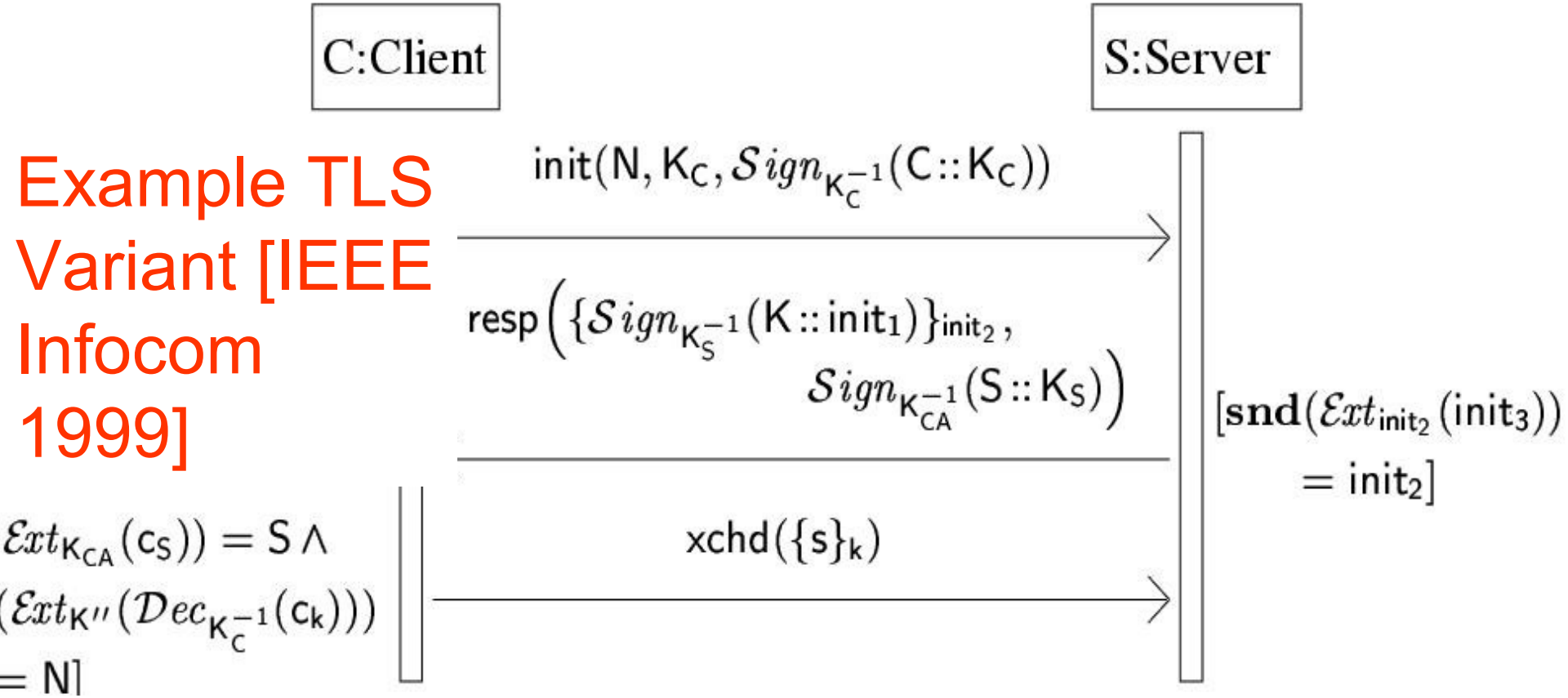
Define cryptosystem etc. E.g.: $Dec_{K^{-1}}(\{E\}_K)=E$

Bound on adversary knowledge set:
Predicate *knows(E)*, means adversary may get to know *E* during the execution of the system.

E.g. secrecy requirement:
For any secret *s*, check whether can derive *knows(s)* from model-generated formulas using automated theorem prover. [ICSE05]

Formal foundations using streams. [JLAP08]

Example TLS Variant [IEEE Infocom 1999]



$$knows(N) \wedge knows(K_C) \wedge knows(Sign_{K_C^{-1}}(C::K_C))$$
$$\wedge\ \forall init_1, init_2, init_3.[knows(init_1) \wedge knows(init_2) \wedge$$
$$knows(init_3) \wedge snd(Ext_{init_2}(init_3)) = init_2$$
$$\Rightarrow knows(\{Sign_{K_S^{-1}}(\ldots)\}_{\ldots}) \wedge [knows(Sign\ldots)]$$
$$\wedge\ \forall resp_1, resp_2.[\ldots \Rightarrow \ldots]]$$

Analysis

Check whether can derive *knows(s)* e.g. using ATP for FOL.

Surprise: Yes !

➔ Protocol does not preserve secrecy of *s*.

Why ? Use Prolog-based attack generator.

```
input_formula(tls_abstract_protocol,axiom,(
 ![ArgS_11, ArgS_12, ArgS_13, ArgC_11, ArgC_12] : (
   ![DataC_KK, DataC_k, DataC_n] : (
      % Client -> Attacker (1. message)
      (       knows(n)
            & knows(k_c)
            & knows(sign(conc(c, k_c),
    & % Server -> Attacker (2. mes
      (  (    knows(ArgS_11)
            & knows(ArgS_12)
            & knows(ArgS_1
            & ( ? [X] :

      => (    kno

    & % C
```

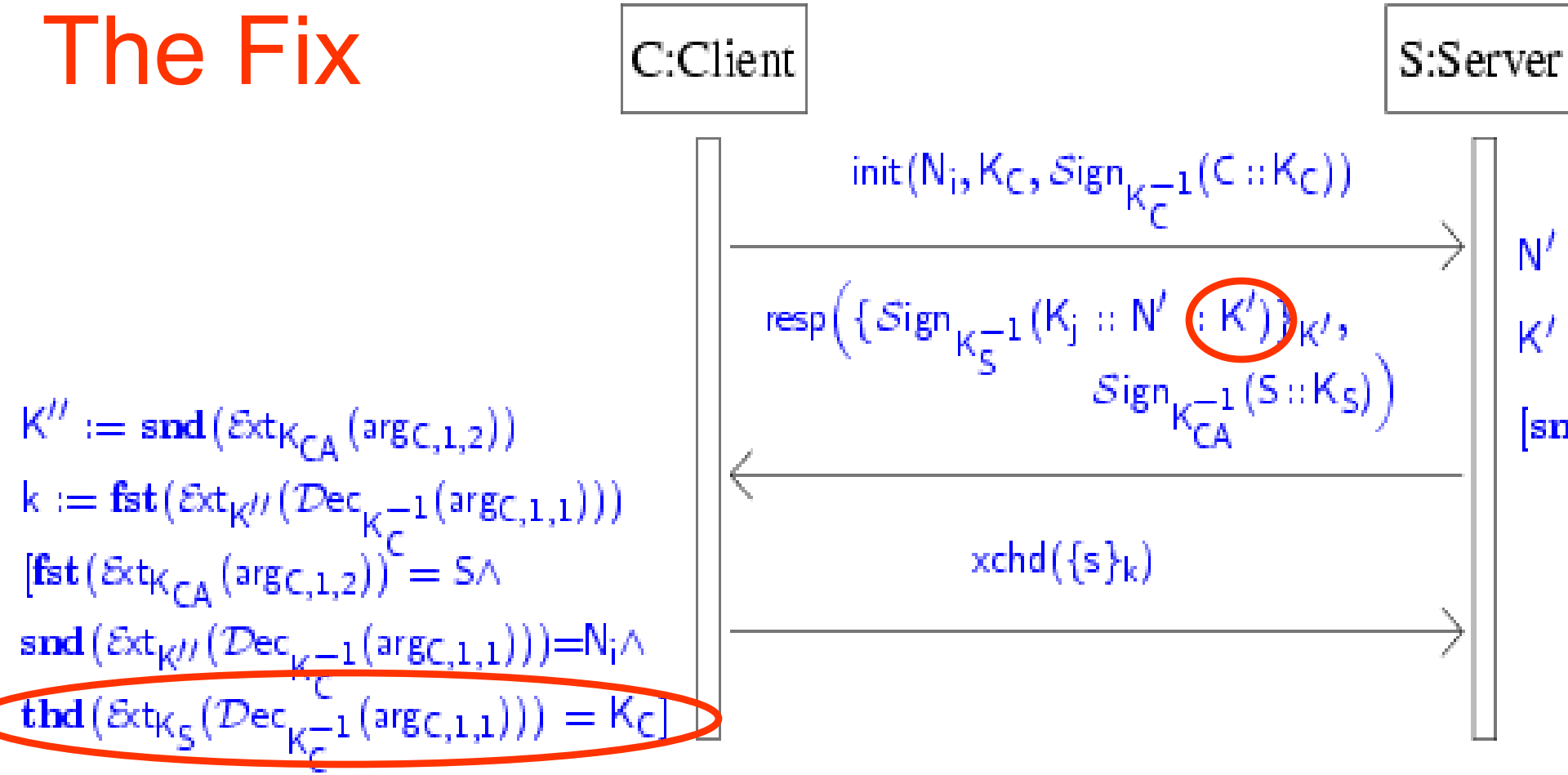
```
E-SETHEO csp03 single processor running on host ...
(c) 2003 Max-Planck-Institut fuer Informatik and
Technische Universitaet Muenchen

analyzing results ...
proof found
time limit information: 298 total / 297 strategy
```

Attack

```
                    , inv(k_ca)), ArgC_1
                    _k, DataC_n), inv(DataC
                    11 )
    & (             equal(sign(conc(s, DataC_ks), i
                          ArgC_12 ) )
    & equal(        gn(conc(DataC_k, n), inv(DataC_KK) )
            ArgC_11 )
    & equal(enc(sign(conc(DataC_k, DataC_n), inv(DataC
            ArgC_11 )
      )
    => ( knows(symenc(secret, DataC_k)) )  )
  ) )  )).
```

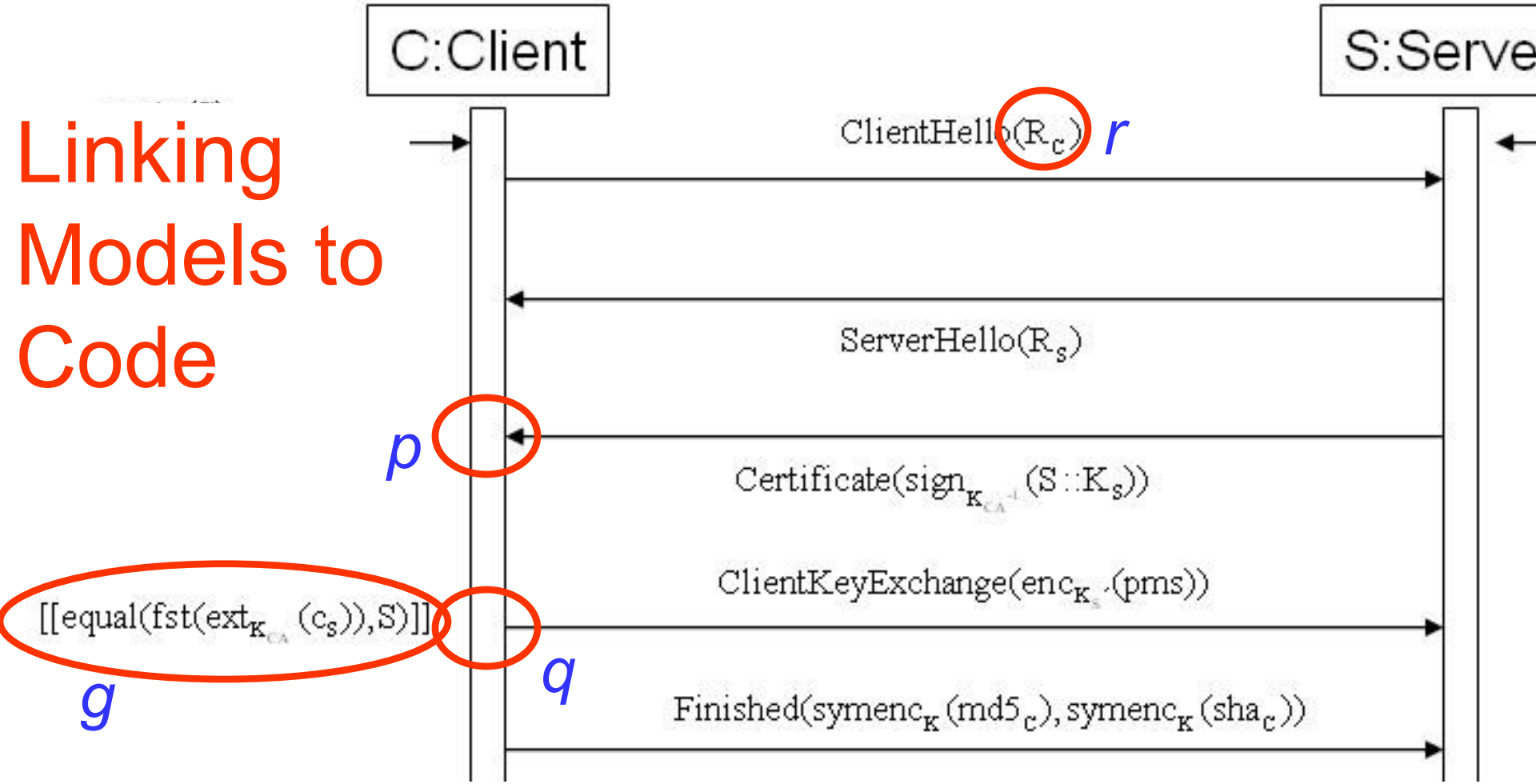
The Fix



e-Setheo: Proof that *knows(s)* not derivable.

Note completeness of FOL (but also undecidability).

Linking Models to Code



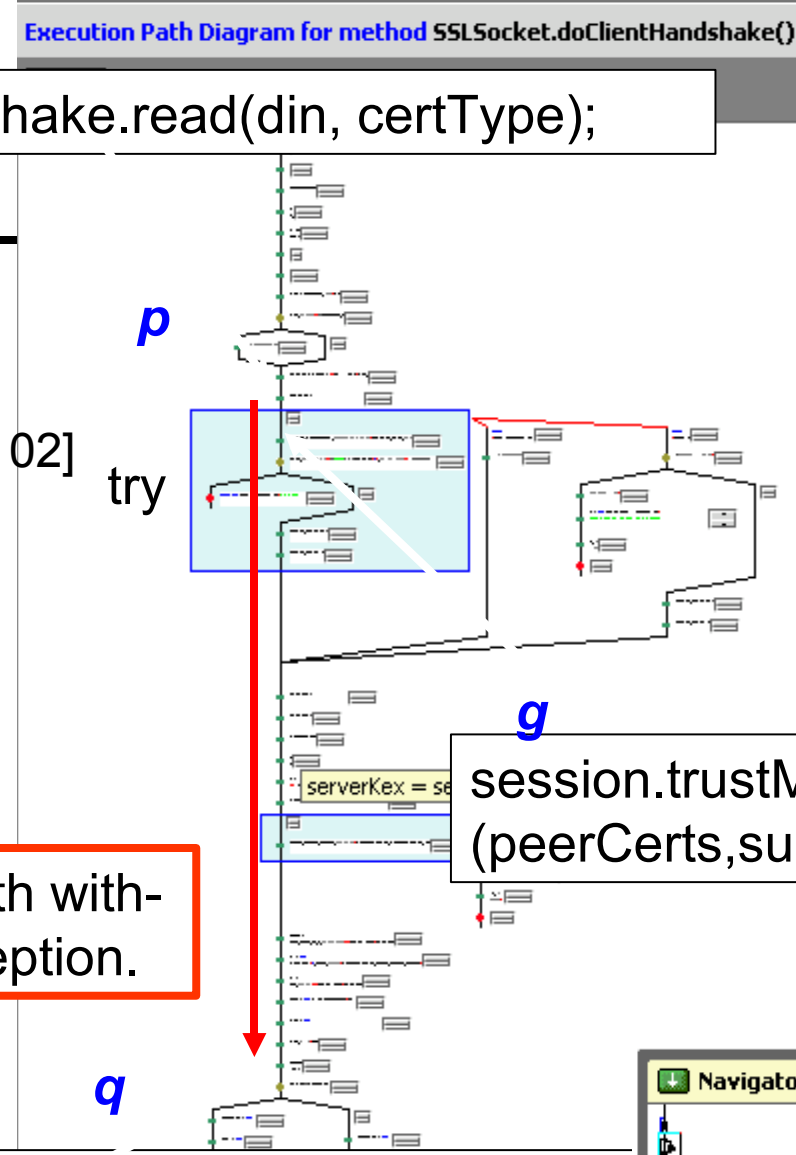
I) Identify program points:

value (*r*), receive (*p*), guard (*g*), send (*q*)

II) Check guards enforced

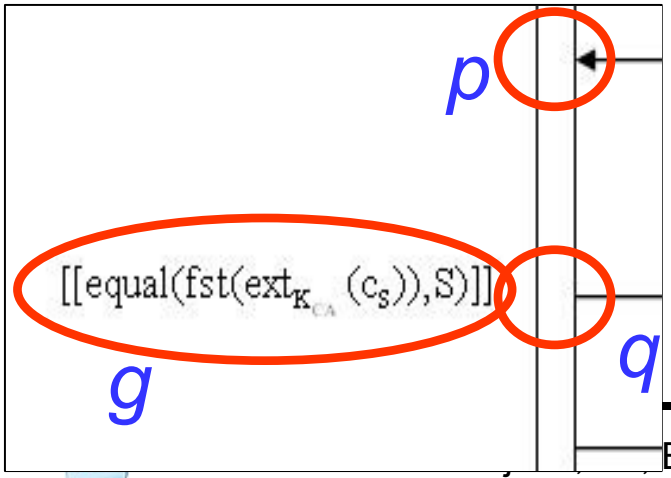
Static Verification

- Guard *g* enforced ?
- Testing (vs. crypto ?) [ASE 01, ICFEM 02]
- Automated formal local verification: Conditionals between *p* and *q* must imply *g*. [ASE 05, ASE 06]



Only path without exception.

msg = new Handshake(Handshake.Type.C
msg.write (dout, version);



Vulnerability in SSL implementation

Analyzed open-source implementation Jessie of SSL protocol.

- According to SSL specification, a certificate with (issuedDate, expiredDate) should be checked whenever a message is received.
- 4 call sites of certificate() were found in the code.
- Only 3 of them call the Veri() function.
- Test cases were constructed to reveal the vulnerability.
- Fix of the vulnerability can be done using AOP techniques.

Another Problem

How do I know the running implementation is still secure after deployment ?

- Does system model capture all relevant aspects about a system ?
- Are assumptions about influences from a system's operational environment reflected adequately ?
- Are the abstractions that need to be made to enable automated static verification of non-trivial systems faithful wrt the verification result ?

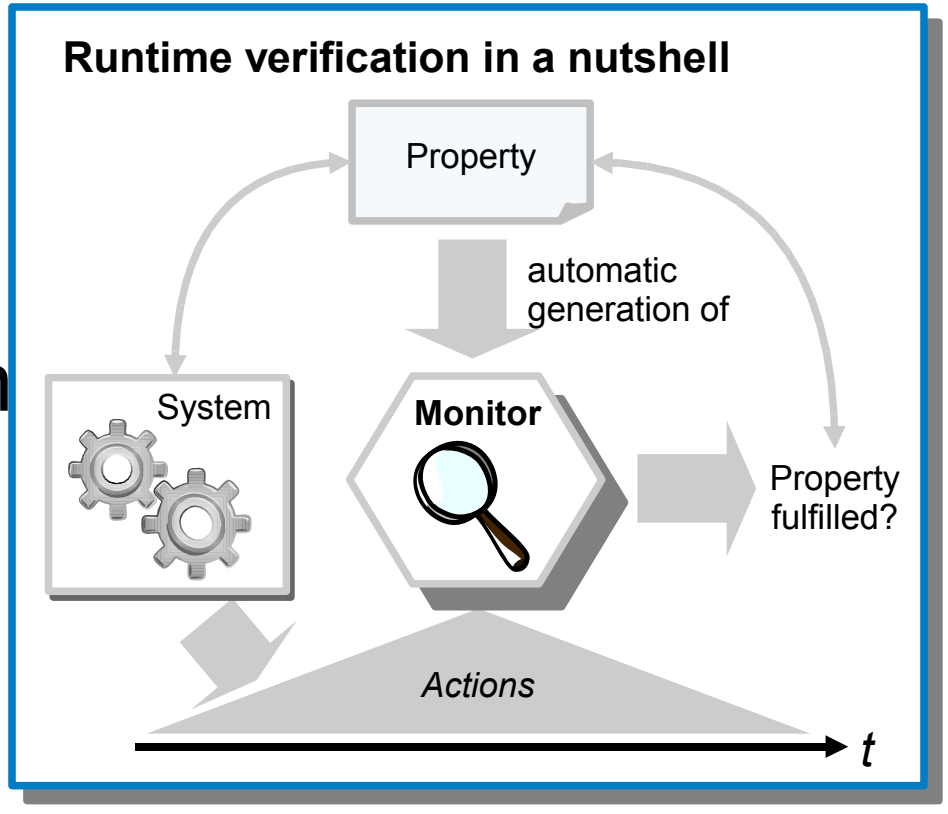
➔ Run-time verification.

Runtime Verification using Monitors

Dynamic verification technique on the actual system.

Essentially a symbiosis of model-checking and testing.

“Lazy model-checking”: only check the system traces which are executed, when they are executed.

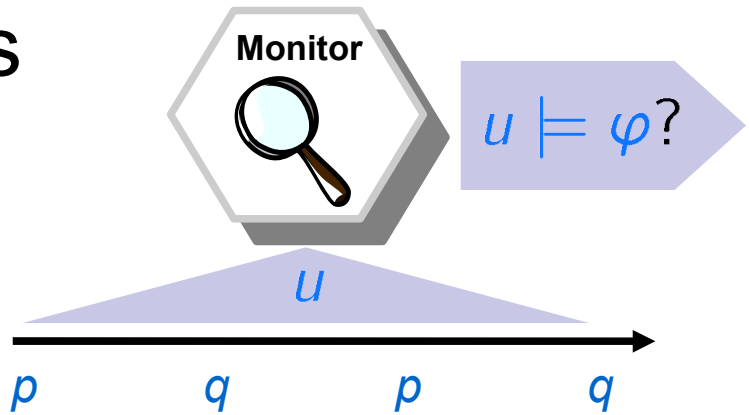


Formal underpinnings

- System (safety) property, φ specified in terms of linear time temporal logic [Pnu77]:

  $\varphi ::= true \mid p \mid \neg p \mid \varphi \; op \; \varphi \mid \varphi \mathbf{U} \varphi \mid \mathbf{X}\varphi \quad (p \in AP)$

- Continuous interpretation of φ over sequence of system events (behaviours), $u \in (2^{AP})^*$



- ***Automatic monitor generation:*** *"Inspired" by translation of LTL to Büchi-automata*

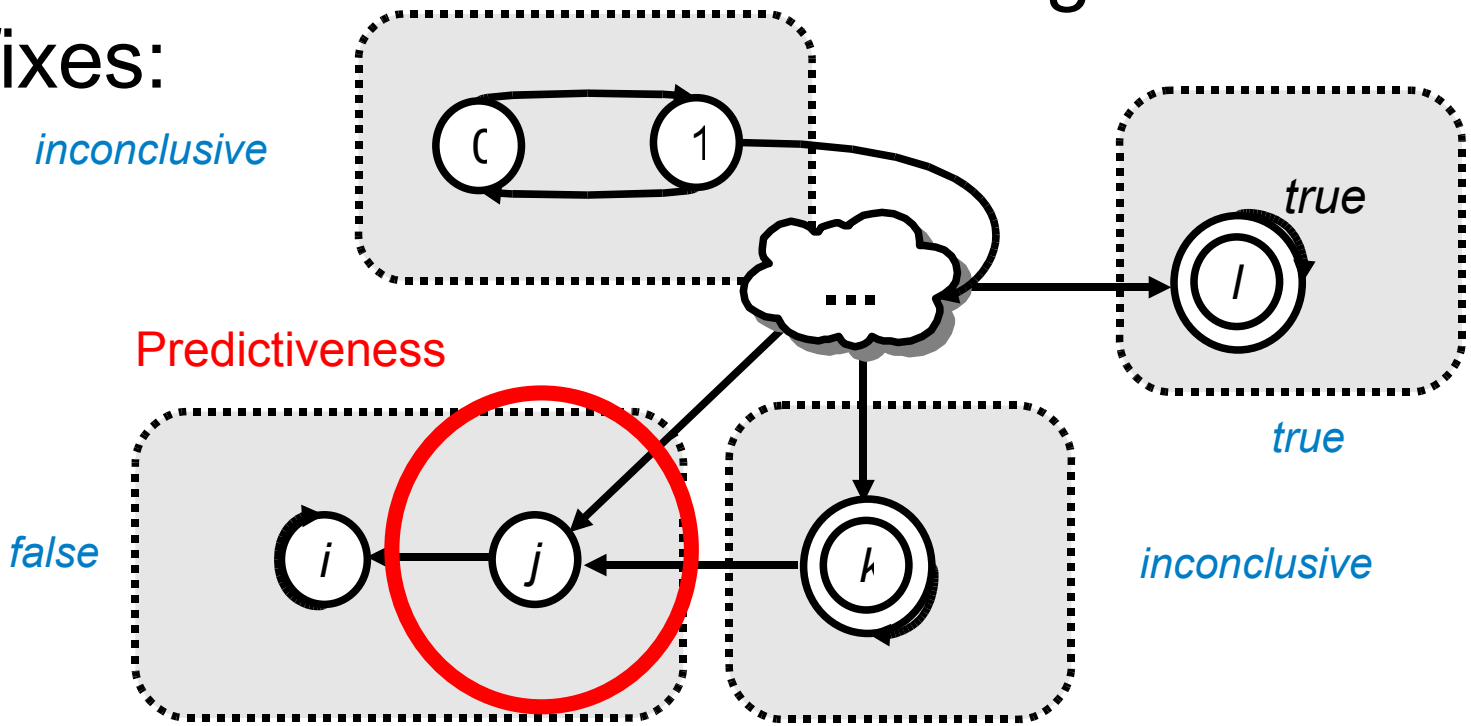
$$\varphi \rightarrow BA_\varphi \; s.t. \; L(BA_\varphi) = L(\varphi)$$

Monitoring-friendly LTL semantics

3-valued semantics:

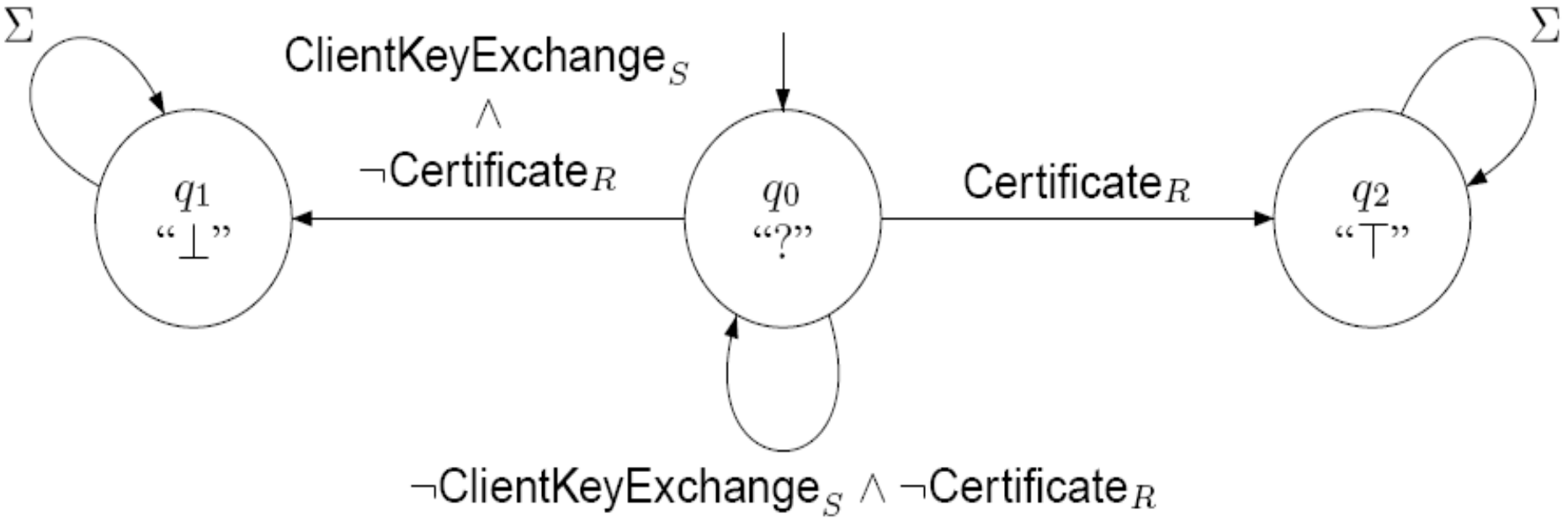
$$[u \models \varphi] = \begin{cases} \top & \text{if } \forall \sigma \in \Sigma^\omega : u\sigma \models \varphi \\ \bot & \text{if } \forall \sigma \in \Sigma^\omega : u\sigma \not\models \varphi \\ ? & \text{otherwise} \end{cases}$$

Gives finite-state machines for detecting *minimal* bad prefixes:



ClientKeyExchange

Client will not send out ClientKeyExchange message until has received Certificate message and check is positive, and then sends it out.



not safety but co-safety

Figure 1: FSM ¬ClientKeyExchange$_S$ UCertificate$_R$.

Client Transport Data

Client will not send any transport data before has checked that MD5 hash received in Server`s Finished message is equal to MD5 created by Client (and correspondingly for SHA hash).

$$\varphi_3 = \neg Data\mathbf{W}((MD5(\mathsf{Finished}_R) = MD5(\mathsf{Finished}_S)),$$

not co-safety but safety

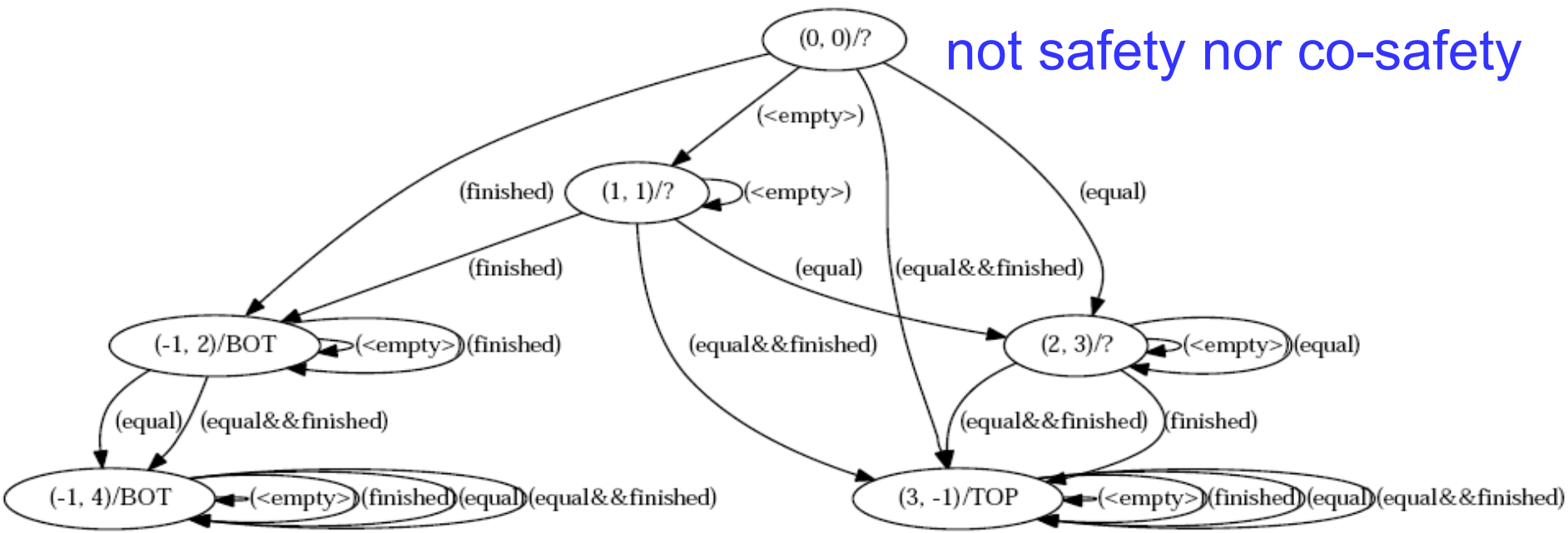
Server Finished

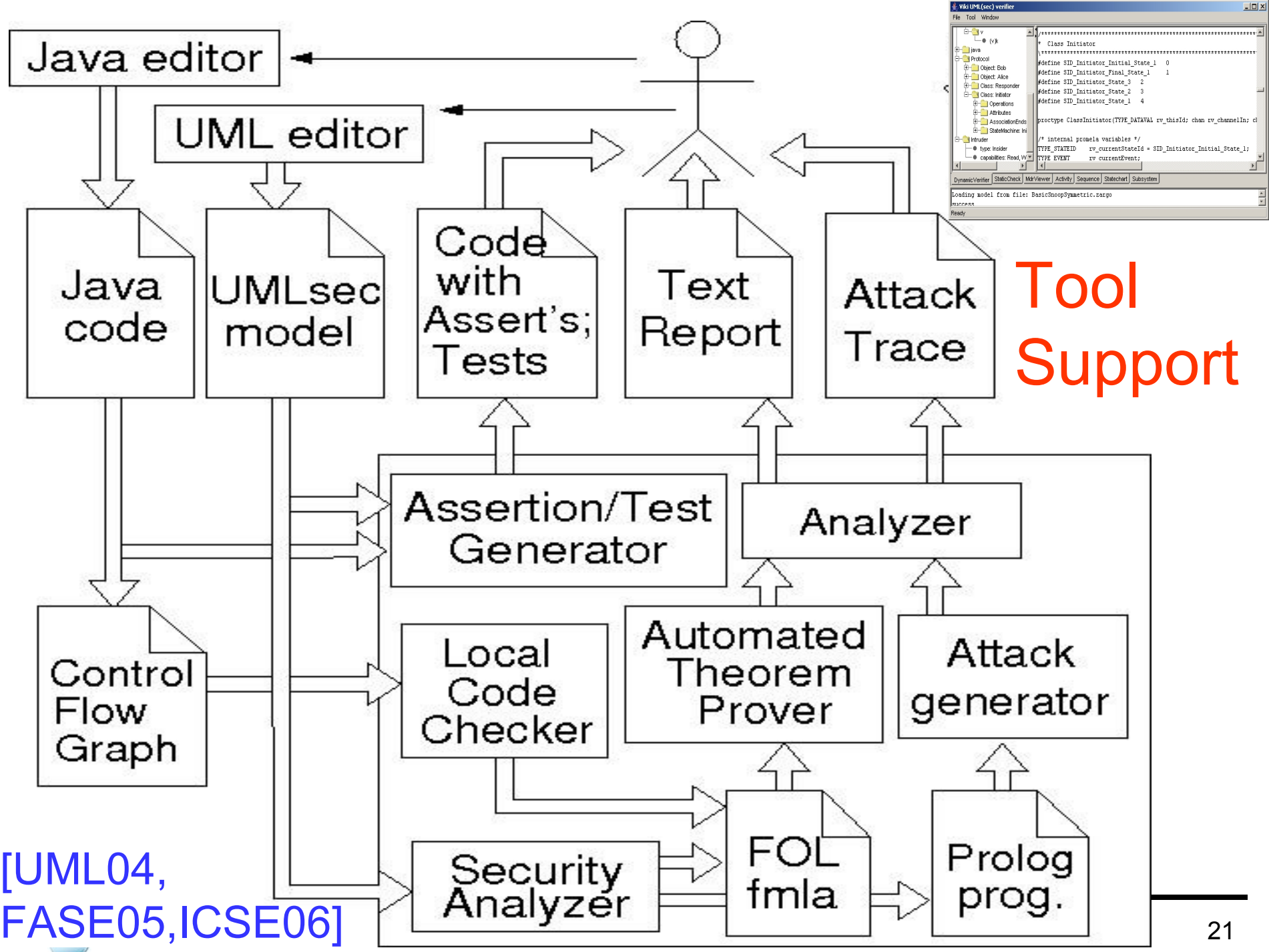
Server will not send Finished message before MD5 received in Client`s Finished message equal to MD5 created by server. Then sends out eventually.

NB: Improves on Schneider's security automata.

$$\varphi_2 = (\neg \text{finished } \mathbf{W} \text{ equal} \wedge (\mathbf{F} \text{ equal} \Rightarrow \mathbf{F} \text{ finished}))$$

not safety nor co-safety




Java editor
UML editor
Java code
UMLsec model
Code with Assert's; Tests
Text Report
Attack Trace
Tool Support
Assertion/Test Generator
Analyzer
Control Flow Graph
Local Code Checker
Automated Theorem Prover
Attack generator
Security Analyzer
FOL fmla
Prolog prog.
[UML04, FASE05,ICSE06]

Some Applications

Analyzed designs / implementations / configurations for

- biometry, smart-card or RFID based identification
- authentication (crypto protocols)
- authorization (user permissions, e.g. SAP systems)

Analyzed security policies, e.g. for privacy regulations.

Conclusion

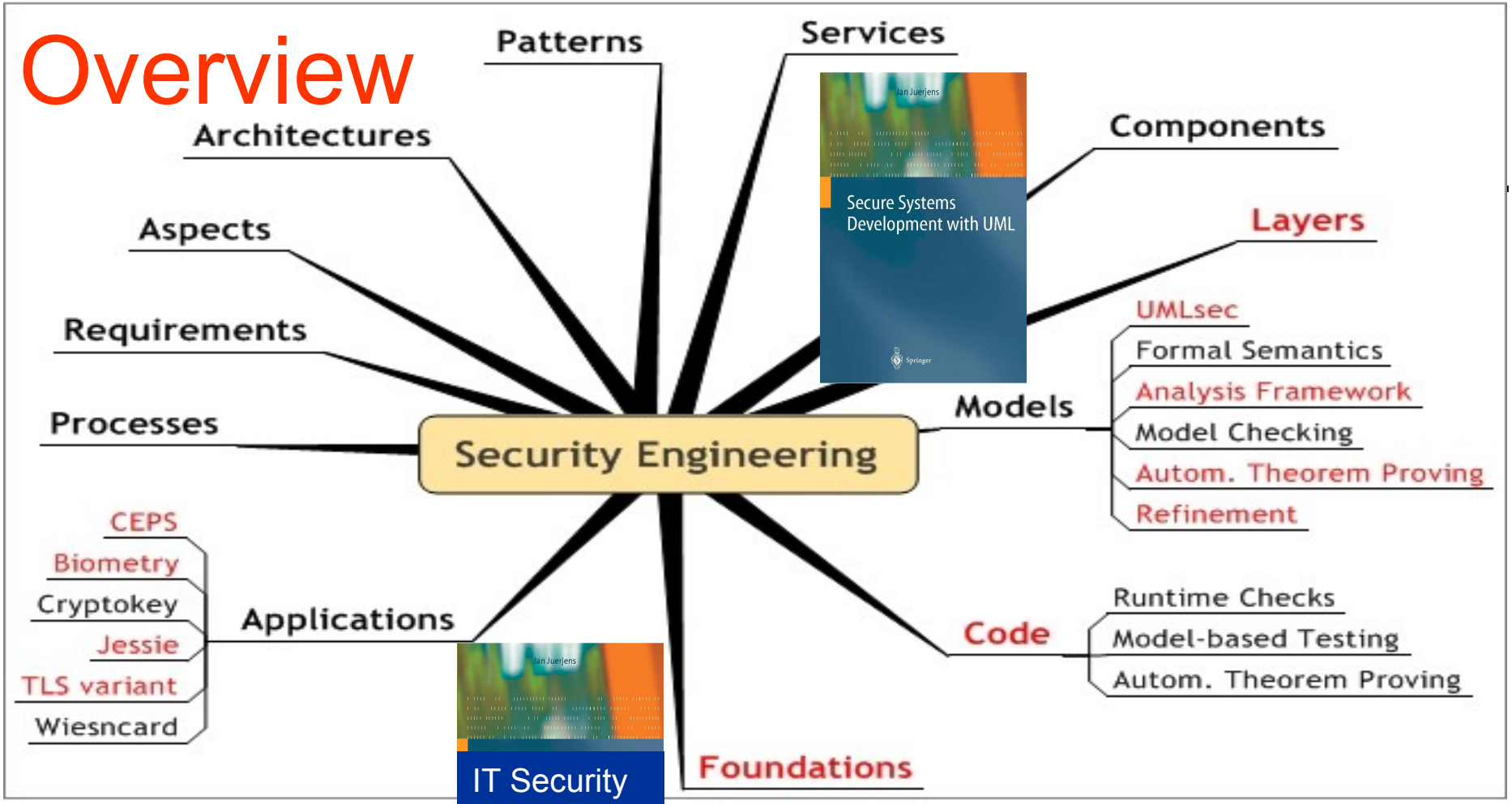
Seemingly first approach to run-time security verification for crypto-based Java implementations.

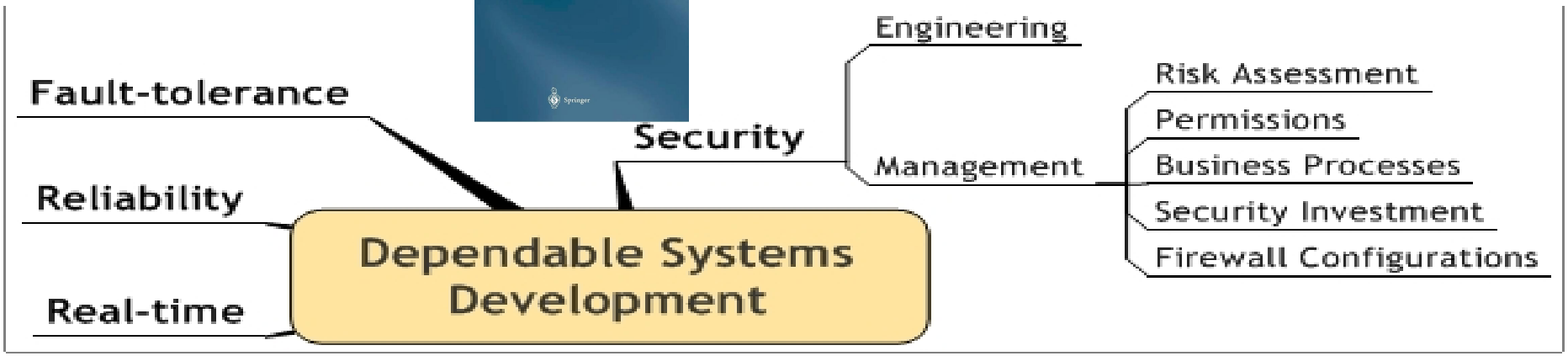
Integrated with static verification of UMLsec models.

Exceeds previous approaches such as Fred Schneider's security automata in expressivity.

Future work: collaboration with Andy Gordon (MSRC) on verifying cryptoprotocol implementations in C.

Overview


Patterns
Services
Architectures
Components
Aspects
Layers
Requirements
UMLsec
Formal Semantics
Analysis Framework
Processes
Models
Security Engineering
Model Checking
Autom. Theorem Proving
Refinement
CEPS
Biometry
Cryptokey
Jessie
TLS variant
Wiesncard
Applications
Runtime Checks
Code
Model-based Testing
Autom. Theorem Proving
Foundations
Secure Systems Development with UML
Jan Juerjens
Springer


Engineering
Fault-tolerance
Risk Assessment
Permissions
Security
Management
Business Processes
Reliability
Security Investment
Firewall Configurations
Dependable Systems Development
Real-time
IT Security
Jan Juerjens
Springer

Questions ?

More information (papers, slides, tool etc.):
http://www.jurjens.de/jan

J.Jurjens@open.ac.uk