Model-based vs. Code-based Verification for Critical Systems

Jan Jürjens

Computing Department,The Open University, GB

from 1 Oct 2008 also:

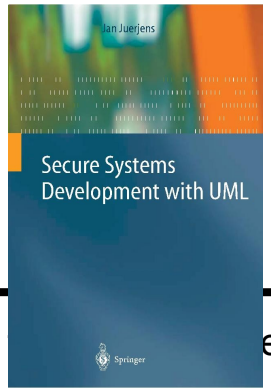
Microsoft Research (Cambridge)

THE ROYAL SOCIETY
CELEBRATING 350 YEARS

http://www.jurjens.de/jan

Personal Introduction

- Senior Lecturer (equiv. US Assoc. Prof.), Computing Departm., The Open University, GB
- From 1 Oct 2008: Royal Society Industrial Fellow at Microsoft Research (Cambridge)
- Extensive collaboration with industry (British Telecom, BMW, HypoVereinsbank, T-Systems, Munich Re, O2, Deutsche Bank, Siemens, Infineon, Allianz, …)
- PhD in Computer Science from Oxford Univ., Masters in Mathematics from Bremen Univ.
- Numerous publicat. inc. 2 books on secure software engineering



Verifying Critical Systems

Very challenging.

For high level of assurance, would need full coverage (test every possible execution).

Usually infeasible (especially reactive systems).

Have heuristics for trade-off between development effort and reliability.

Need to ask yourself:

- How complete is the heuristic ?
- How can I validate it ?

This talk: focus on security. Generalizes to other criticality requirements (fault-tolerance, reliability, …)

Problem: Security is Elusive

- Classical weakness in old Unix systems: “wrong password” message at first wrong letter in password. Using timing attack, reduce password space from 26^n to 26*n (n = password length)
- More recent weakness on smart-card: reconstruct secret key by timed measurement of power consumption during crypto operations

➔ How do you find these weaknesses using classical testing ?

(Yon don’t.)

Problem: Untrustworthy Programmer

- For security assurance, may not even trust the programmer of the code.
- May have intentionally built in back-door into code.
- May be impossible to find by random or black-box testing (e.g. hard-coded special password).
- Even worse when elusive weaknesses are used (previous slide).

➔ What is the precaution in practice?

(Usually none.)

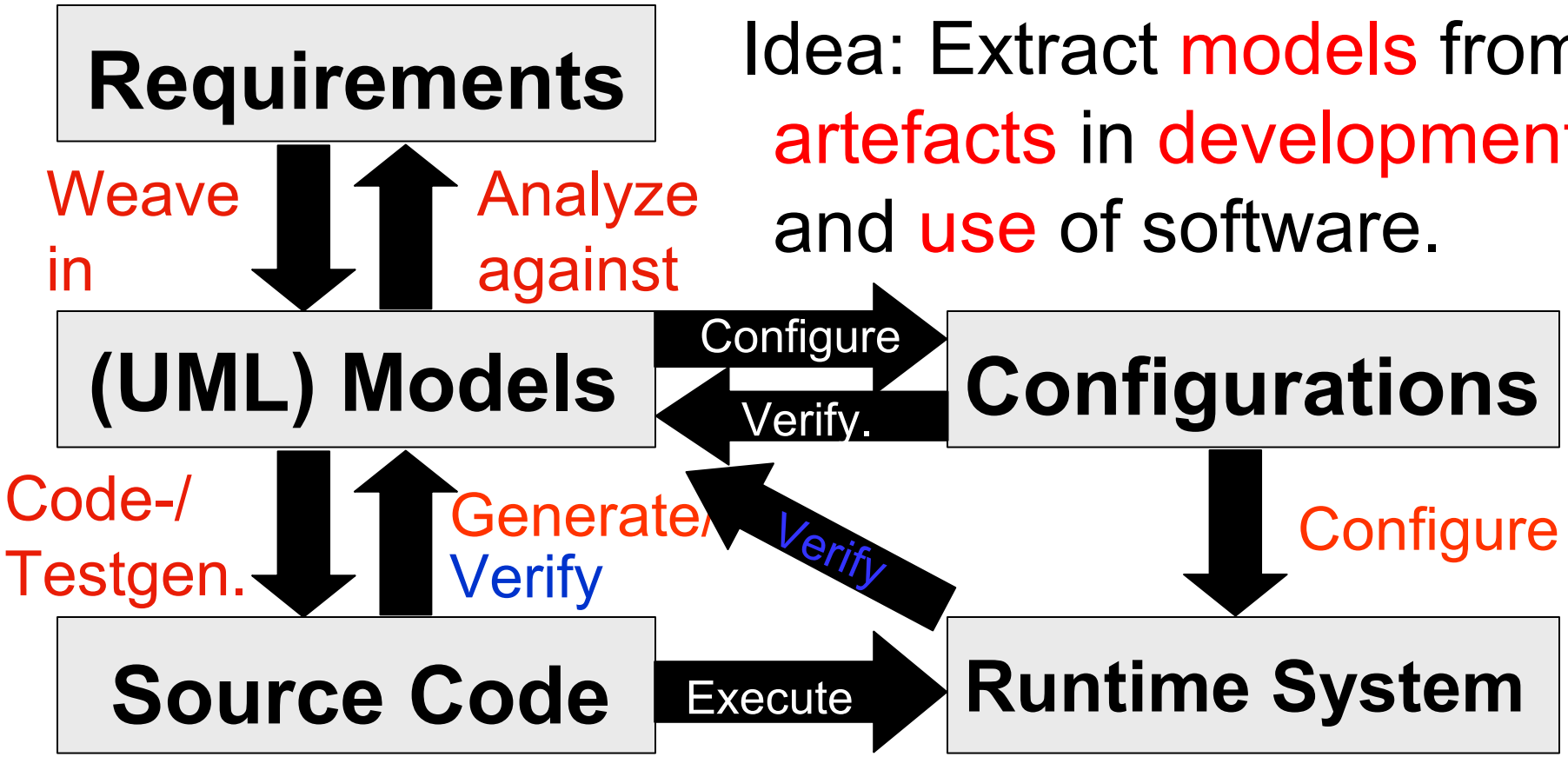
Special Problem: Crypto

- Cryptography plays important role in many security-critical applications
- By definition, needs to be secure against brute-force attacks

➔ Paradox: How do you get sufficient test coverage (for inputs accessible to a given attacker) of a system that needs to be secure against brute-force attacks on that input ?

(Not using classical testing.)

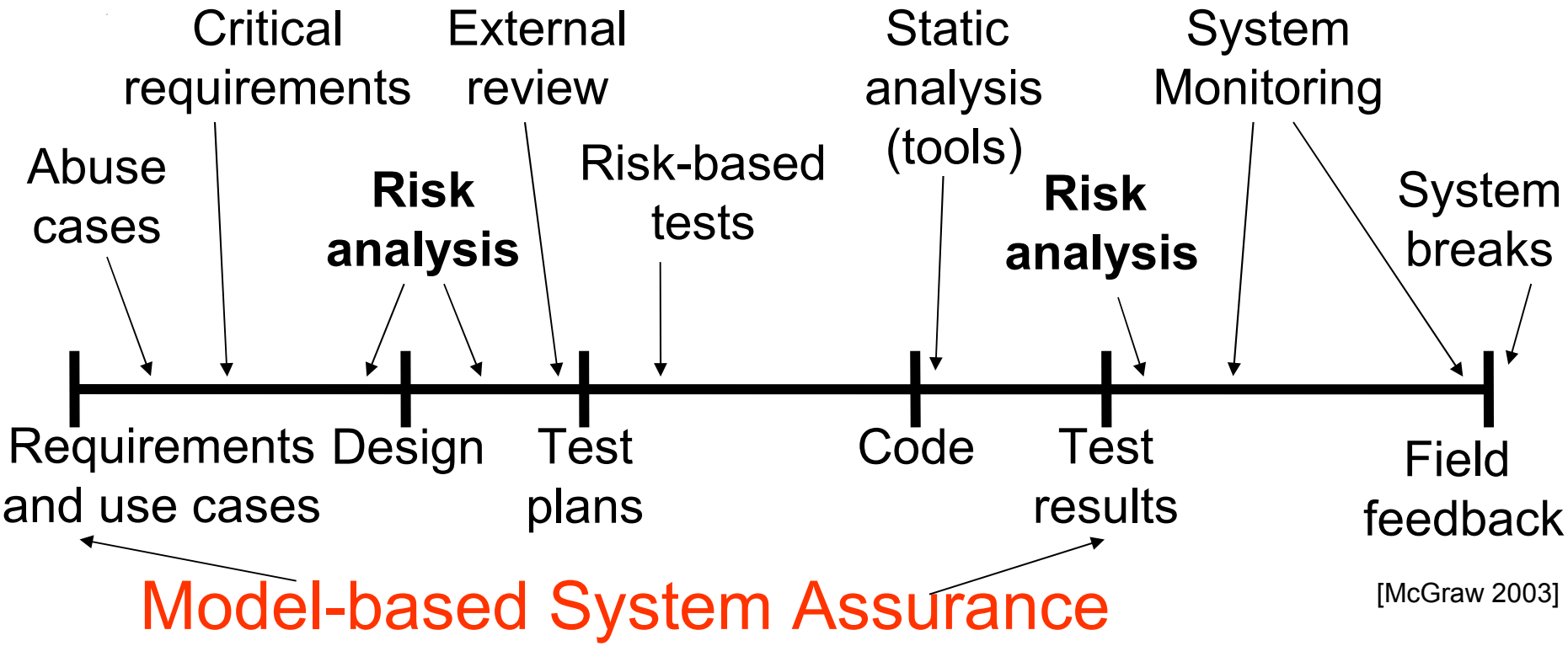
Model-based System Assurance



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

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

Critical System Lifecycle

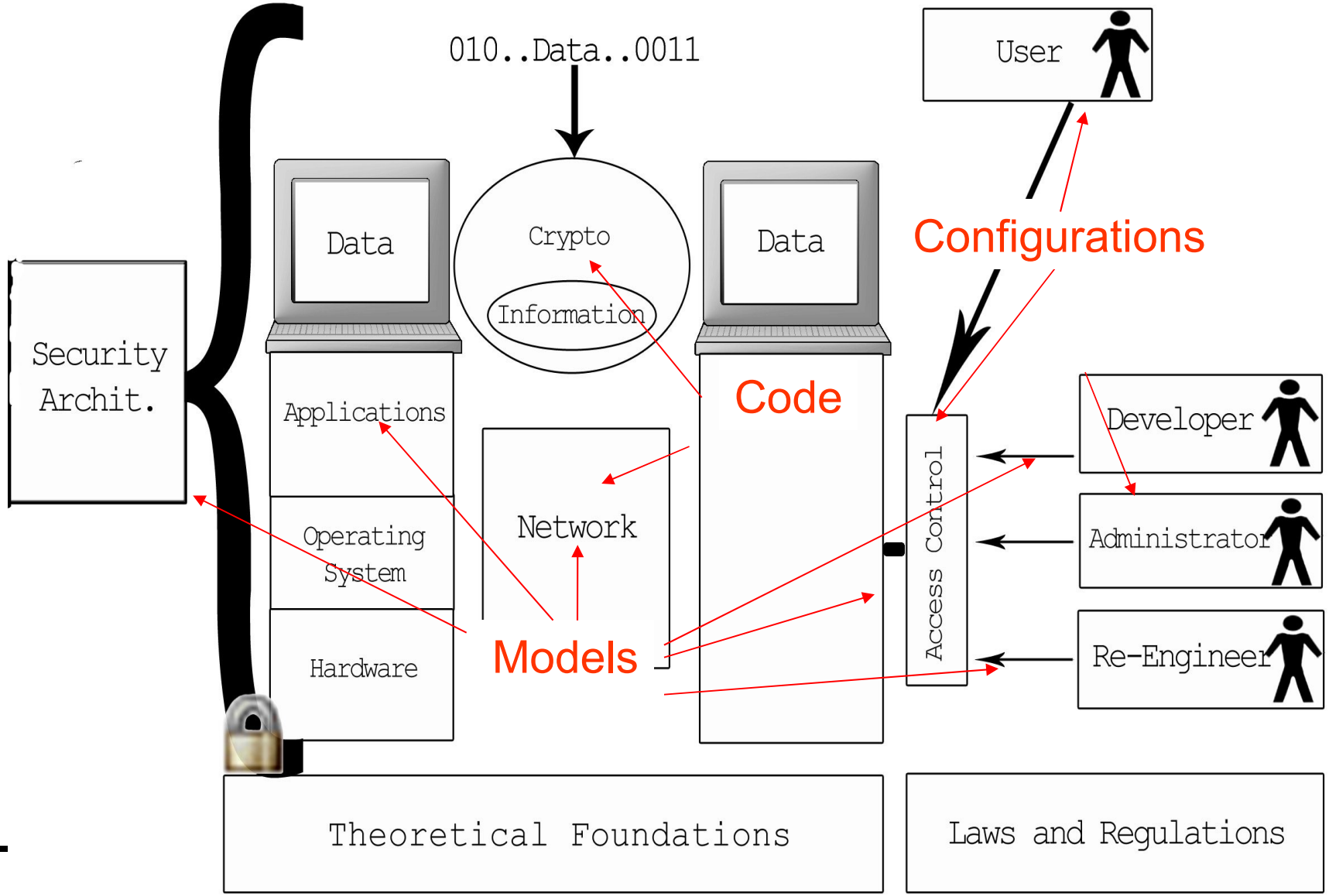


Model-based System Assurance

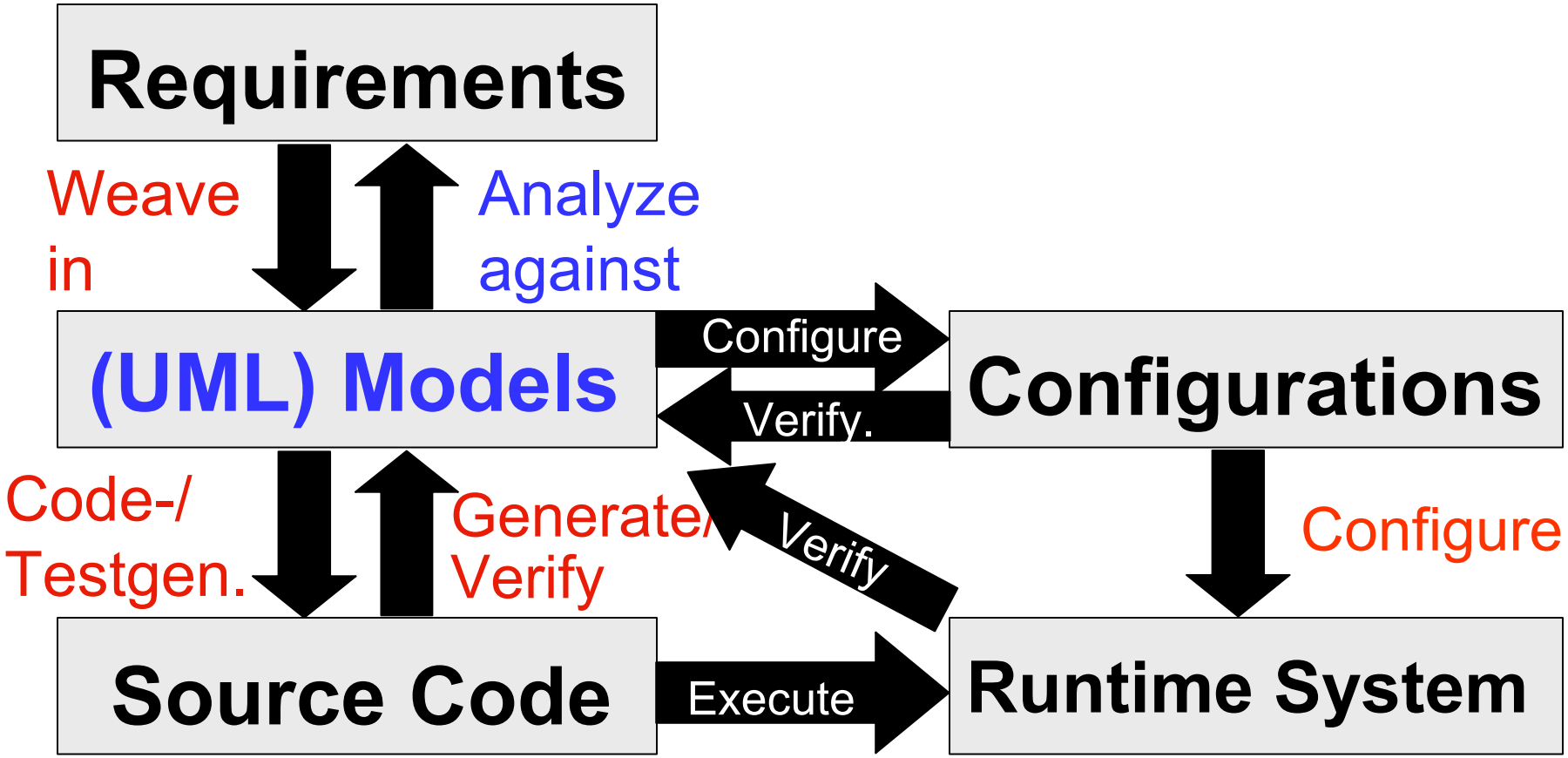
[McGraw 2003]

Design: Encapsulate prudent engineering rules.
Analysis: Formally based, automated, efficient tools.
Note: emphasis on high-level requirements.

Architectural Layers



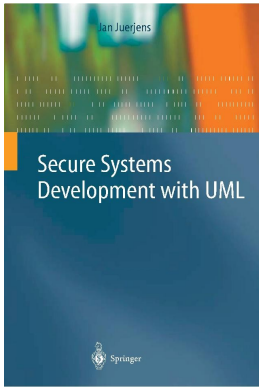
Roadmap


Requirements
Weave in
Analyze against
(UML) Models
Configure
Verify.
Configurations
Code-/ Testgen.
Generate/ Verify
Verify
Configure
Source Code
Execute
Runtime System

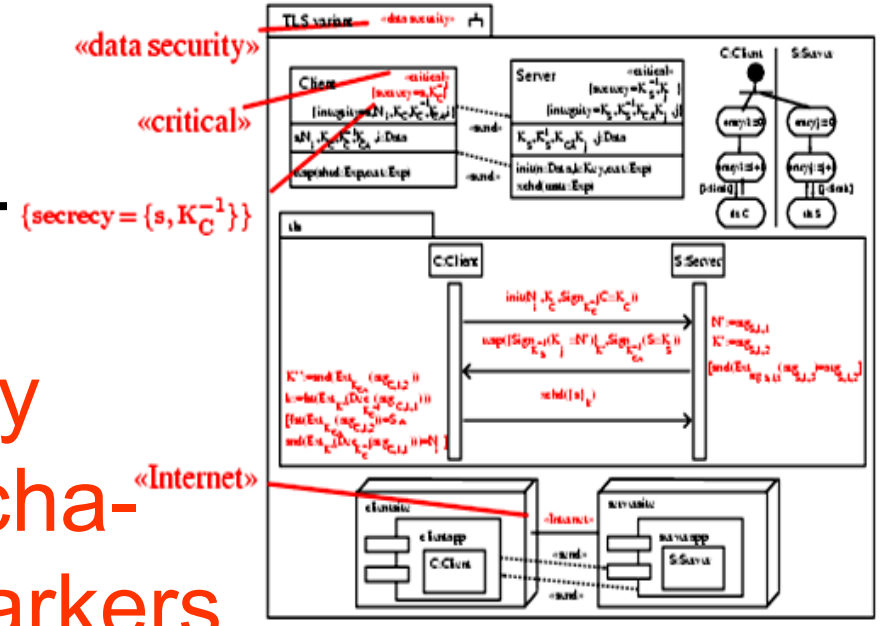
Model-based Security with UMLsec

Extension of the Unified Modeling Language (UML) for secure systems development.

- evaluate UML models for security
- encapsulate established rules of prudent secure engineering
- make available to developers not specialized in secure systems
- consider security requirements from early design phases, in system context
- can use in certification



UMLsec



Insert recurring security requirements, adversary scenarios, security mecha-nisms as predefined markers.

Use associated logical constraints to verify specifications using model checkers and ATPs based on formal semantics.

Ensures that UML specification enforces the relevant security requirements wrt Dolev-Yao type adversaries. [FASE01,UML02,FOSAD05,ICSE05]

What Does UMLsec Cover ?

Security requirements: <<secrecy>>,…

Threat scenarios: Use $\text{Threats}_{\text{adv}}$(ster).

Security concepts: For example <<smart card>>.

Security mechanisms: E.g. <<guarded access>>.

Security primitives: Encryption built in.

Physical security: Given in deployment diagrams.

Security management: Use activity diagrams.

Technology specific: Java, CORBA security.

Security Protocols

System distributed over untrusted networks.

„Adversary“ intercepts, modifies, deletes, inserts messages.

Cryptography provides security.

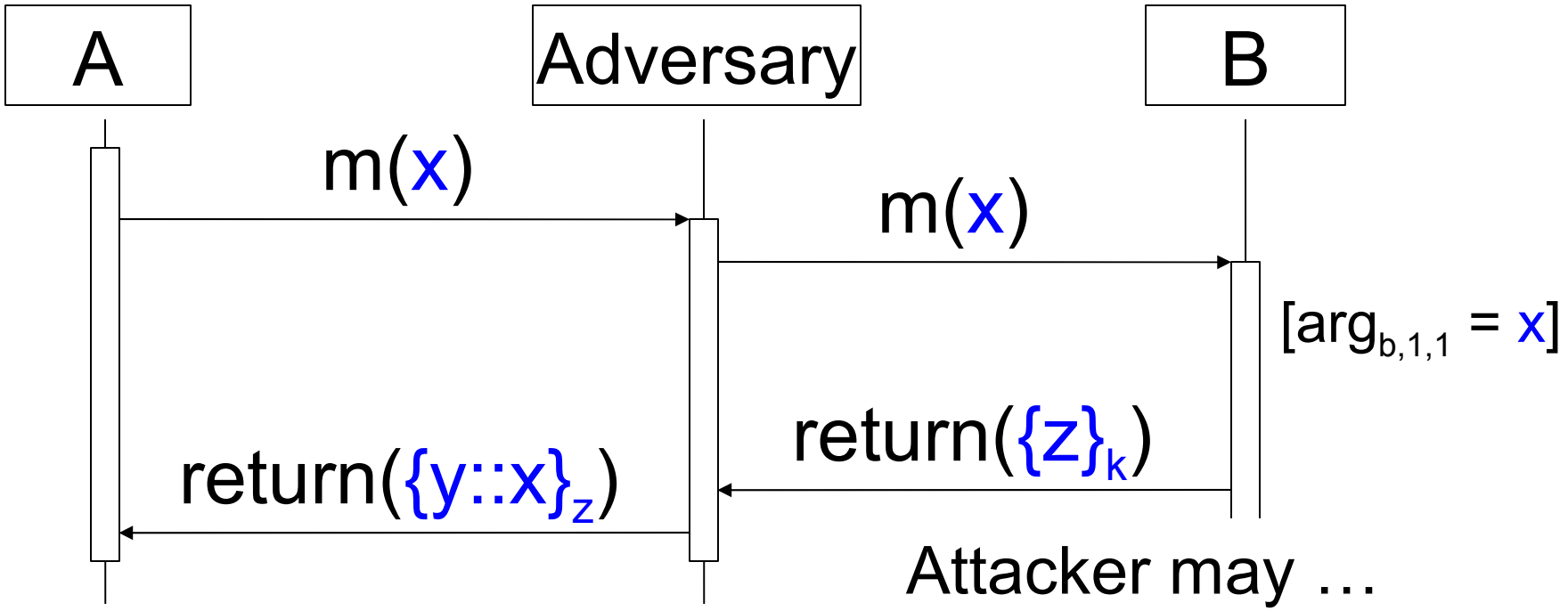
Cryptographic Protocol: Exchange of messages for distributing session keys, authenticating principals etc. using cryptographic algorithms

Security Protocols: Problems

Many protocols have vulnerabilities or subtleties for various reasons

- weak cryptography
- core message exchange
- interfaces, prologues, epilogues
- deployment
- implementation bugs

Crypto-based Software (e.g. Protocols)



Adversary knowledge: k^{-1}, y, x, $\{z\}_k$, z

(cf. [Dolev, Yao 1982])

Attacker may …

- control system parts,
- know data in advance,
- intercept messages,
- delete messages,
- inject messages.

Example: TLS Variant

«data security»

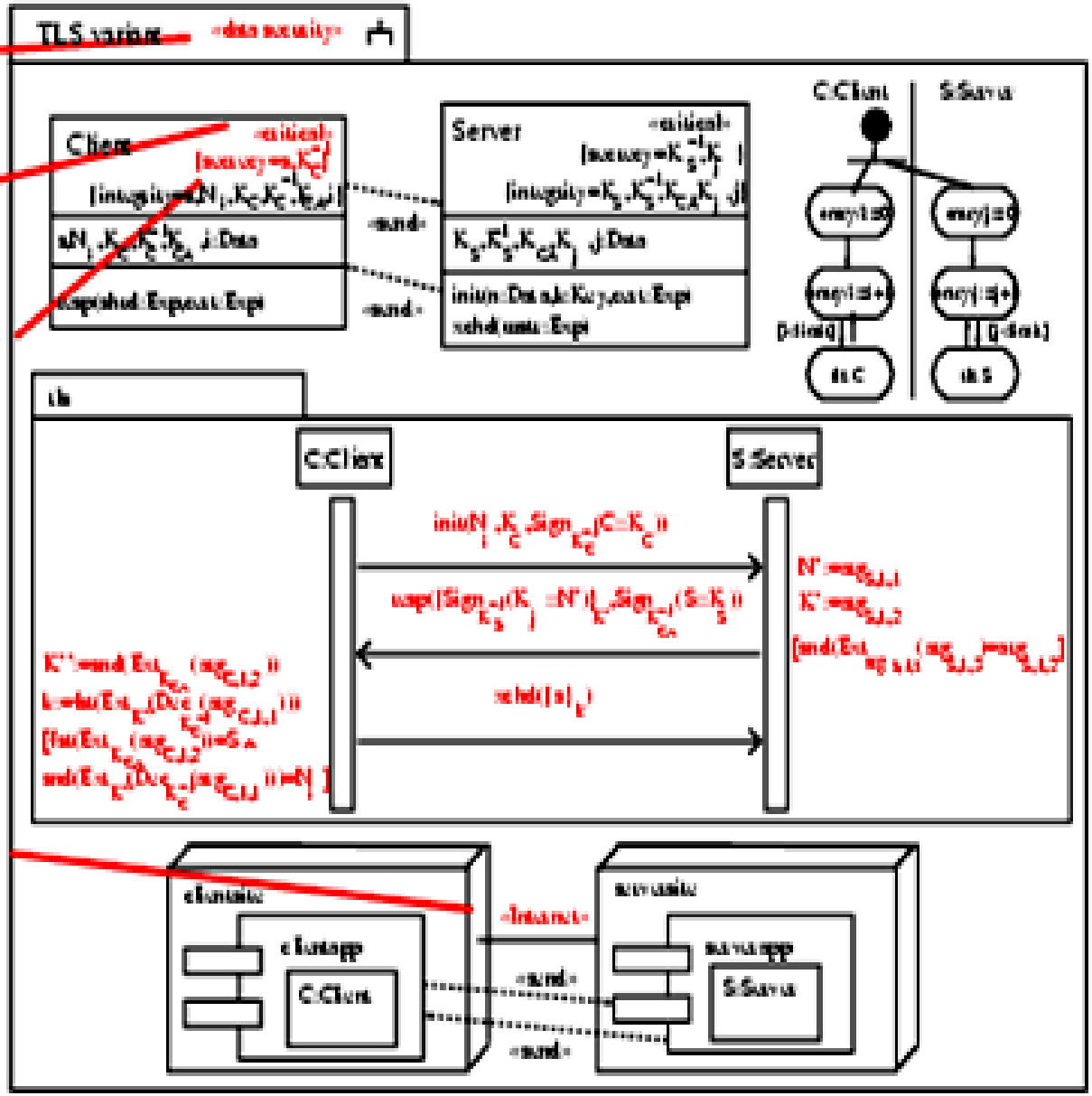
Presented at IEEE Infocom 1999

«critical»

$\{secrecy = \{s, K_C^{-1}\}\}$

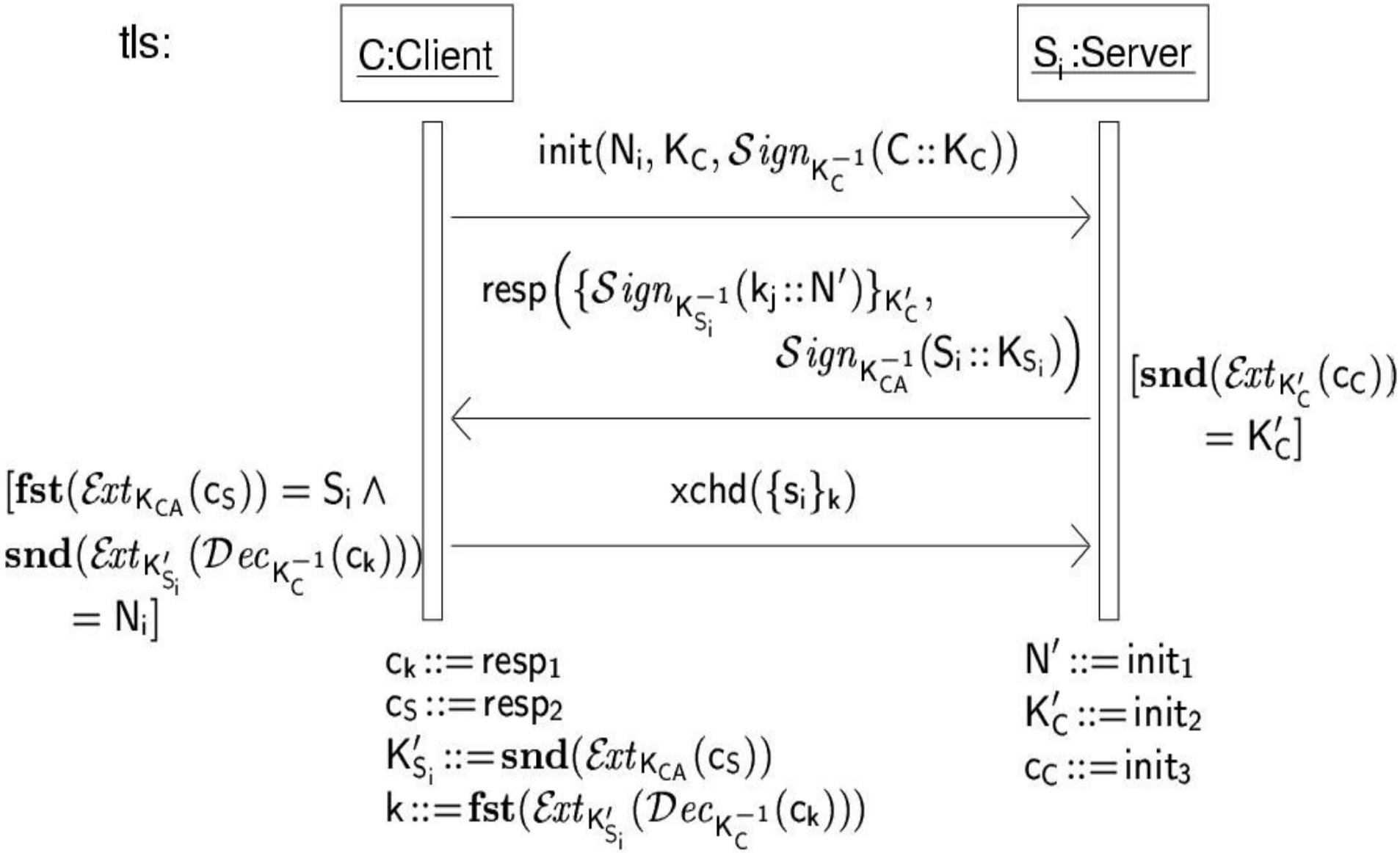
Goal: send secret protected by session key using fewer server resources.

«Internet»



Protocol

tls:



Security Analysis in First-order Logic

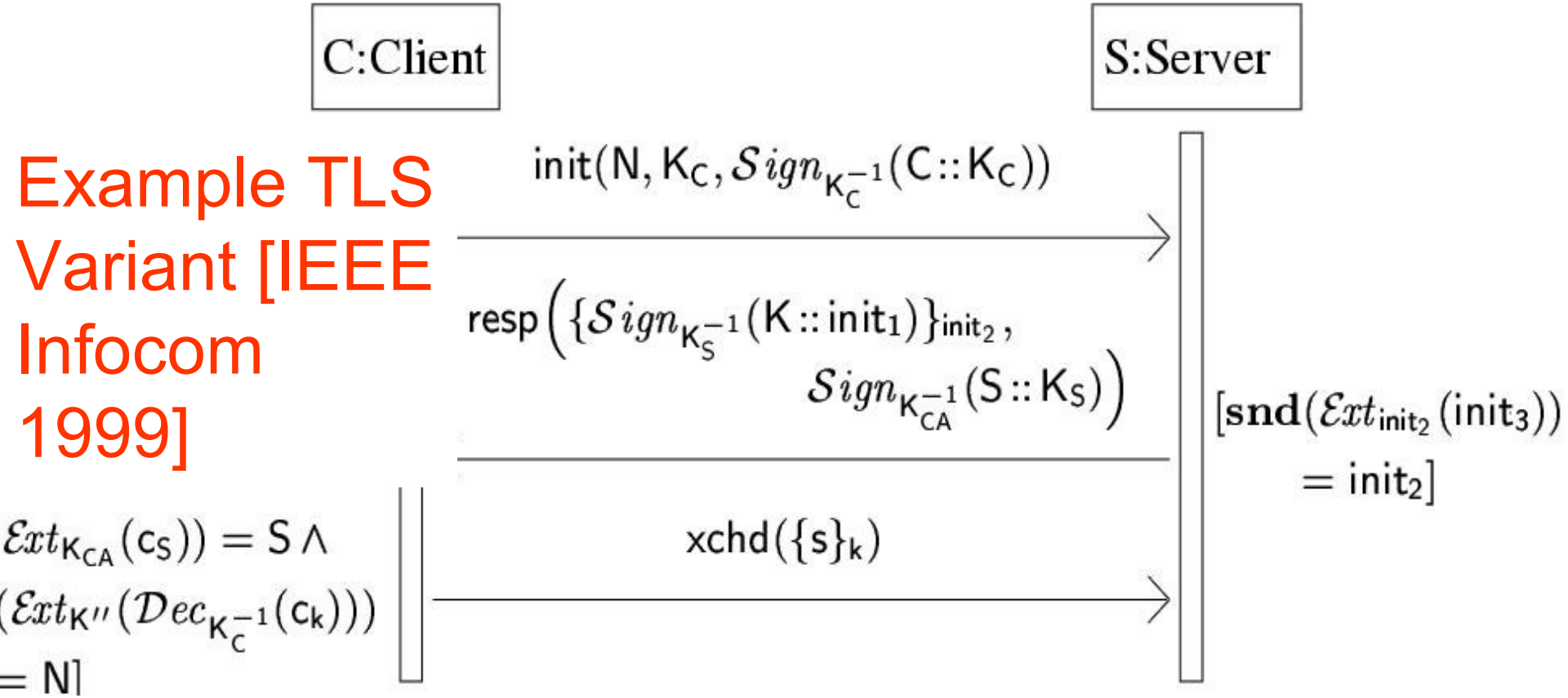
Define cryptosystem etc. E.g.: $Dec_{K^{-1}}(\{E\}_K)=E$

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), inv(k_c) ) )  )
    & % Server -> Attacker (2. message)
      (  (    knows(ArgS_11)
            & knows(ArgS_12)
            & knows(ArgS_13)
            & ( ? [X] : equal(                                  inv(A
                                                  schedule ...
                                                  ...
                                                  entering next
                                                  ...
                                                  analyzing results ...
                                                  proof found
                                                  time limit information: 29
                                                  ...
                                                  e-SETHEO done. exiting
      => (    knows(enc(
                                  ArgS_
            & knows(sign(conc(s,                        (k_ca) ) ) )  )
   & % Client -> Attacker (3. mes
     (  (    knows(ArgC_11)
           & knows(ArgC_12)
           & equal(sign(conc(s, DataC_KK), inv(k_ca)), ArgC_1
           & equal(enc(sign(conc(DataC_k, DataC_n), inv(DataC
                       k_c), ArgC_11 )
           & ( ? [DataC_ks] : equal(sign(conc(s, DataC_ks), i
                                    ArgC_12 ) )
           & equal(enc(sign(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)) )  )
  ) )  )).
```

Man-in-the-Middle Attack

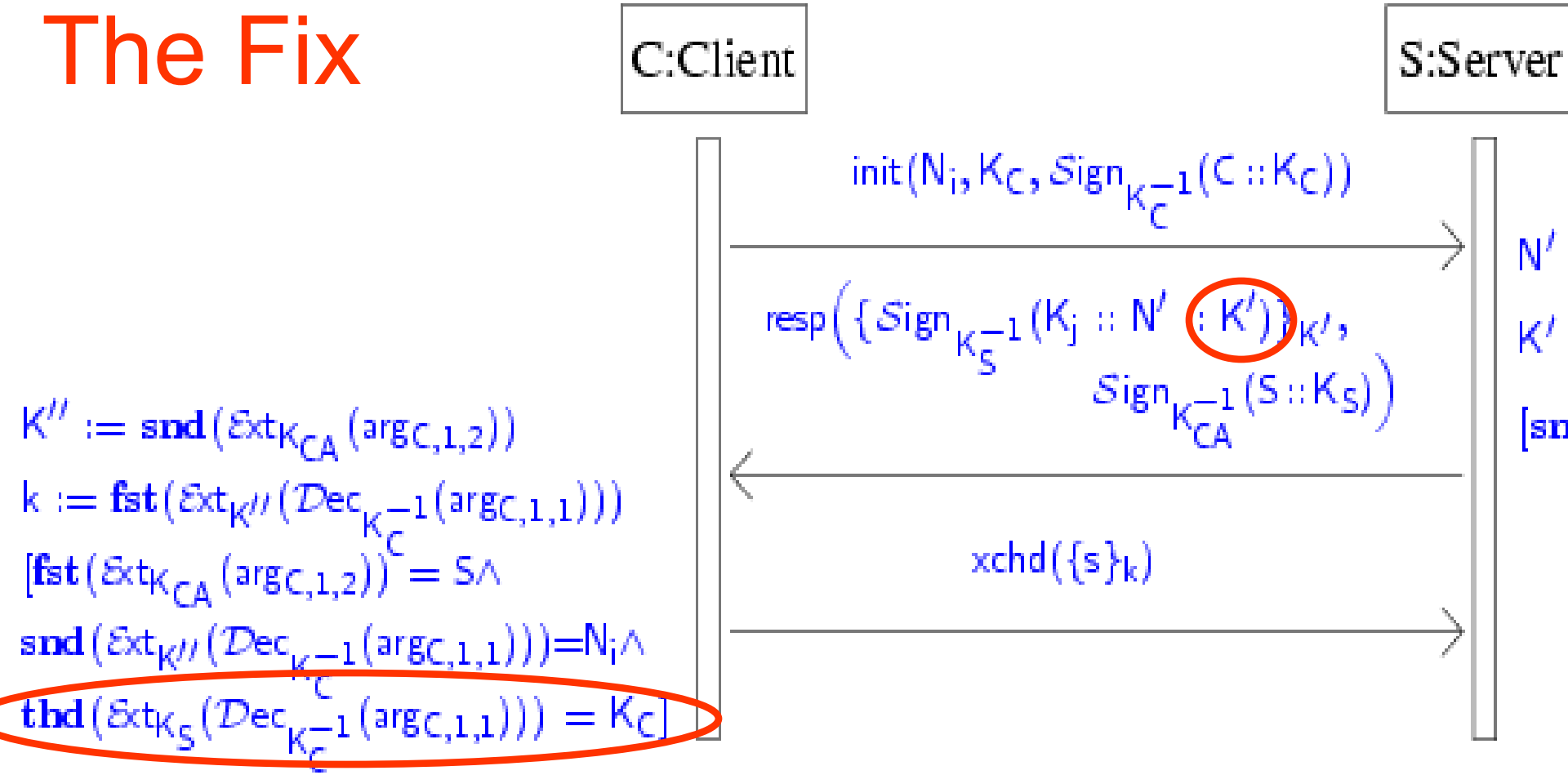
$$C \xrightarrow{N_i::K_C::Sign_{K_C^{-1}}(C::K_C)} A \xrightarrow{N_i::K_A::Sign_{K_A^{-1}}(C::K_A)} S$$

$$A \xleftarrow{\{Sign_{K_S^{-1}}(K_j::N_i)\}_{K_A}::Sign_{K_{CA}^{-1}}(S::K_S)} S$$

$$C \xleftarrow{\{Sign_{K_S^{-1}}(K_j::N_i)\}_{K_C}::Sign_{K_{CA}^{-1}}(S::K_S)} A$$

$$C \xrightarrow{\{s\}_{K_j}} A \xrightarrow{\{s\}_{K_j}} S$$

The Fix



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

Note completeness of FOL (but also undecidability).

Refinement, Composability, Aspects, Services

Need to refine models down to code.

Common formalizations of security properties not preserved by refinement.

Bad: re-verify after each step (incl code).

Theorem: Our notion of model refinement preserves security requirements. [FME01]

Similar: Established composability for certain security requirements under suitable assumptions. [Concur01]

Also: Demonstrated how to apply security using aspect-oriented weaving / service orientation. [ICSOC 04, Models 05]

Layered Security Protocols

System layer on top uses security services below.

client authenticity

confidentiality, integrity, server authenticity

=

confidentiality, … + client authenticity

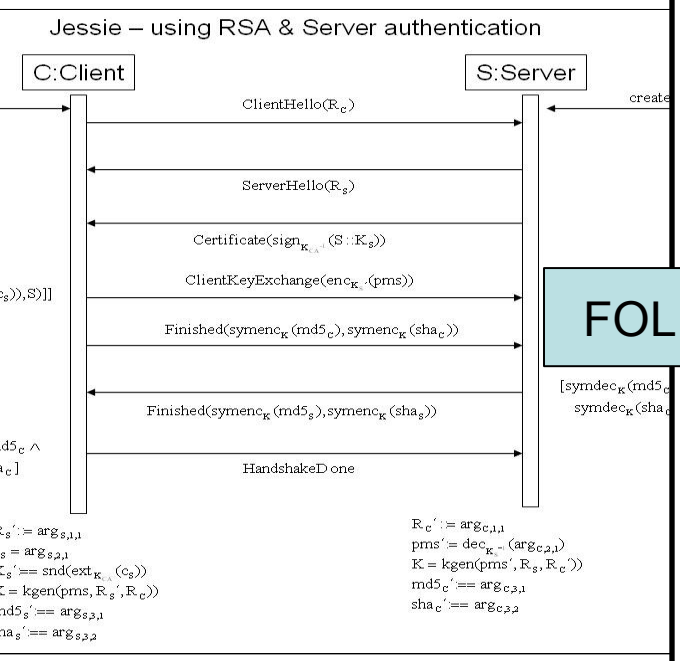
?

Security properties additive ? [Safecomp03]

Theorem: Yes, under suitable conditions.

Model Verification

$$\forall arg_1, \ldots, arg_n \;.\; (knows(arg_1) \wedge \ldots \wedge knows(arg_n) \wedge cond(arg_1, \ldots, arg_n) \Rightarrow knows(exp(arg_1, \ldots, arg_n)))$$



FOL

```
...
((
  knows(ArgC_3)
  & (equal(fst(ArgC_3), type_serverkeyexchange))
  & (equal(snd(ext(snd(snd(ArgC_3)), k_ca)), skey))
  & (equal(snd(ext(snd(ArgC_2), k_ca)), fst(snd(ArgC_3))))
)
=>(
  ((knows(ArgC_4_1)
    & equal(ArgC_4_1, type_serverhellodone))
    =>(
  ( ( true & equal(ClientKeyExchange, enc(premasterkey, skey))
)
...
%------------------------- Conjecture --

input_formula(attack,conjecture,(
   knows(mastersecret) )).
```

ATP

```
analyzing results ...
model found/total failure
time limit information: 19 total / 18 strategy
(leaving wrapper).
task myUML_PID1491 on atbroy1 has status SUCCESS
(model found by strategy 300) consuming 1 seconds
deleting temporary files.
e-SETHEO done. exiting
```

[FASE05,ICSE05, ICSE06]

Tool-support: Pragmatics

Commercial modelling tools: so far mainly syntactic checks and code-generation.

Goal: sophisticated analysis. Solution:

- Draw UML models with editor.
- Save UML models as XMI (XML dialect).
- Connect to verification tools (automated theorem prover, model-checker …), e.g. using XMI Data Binding.

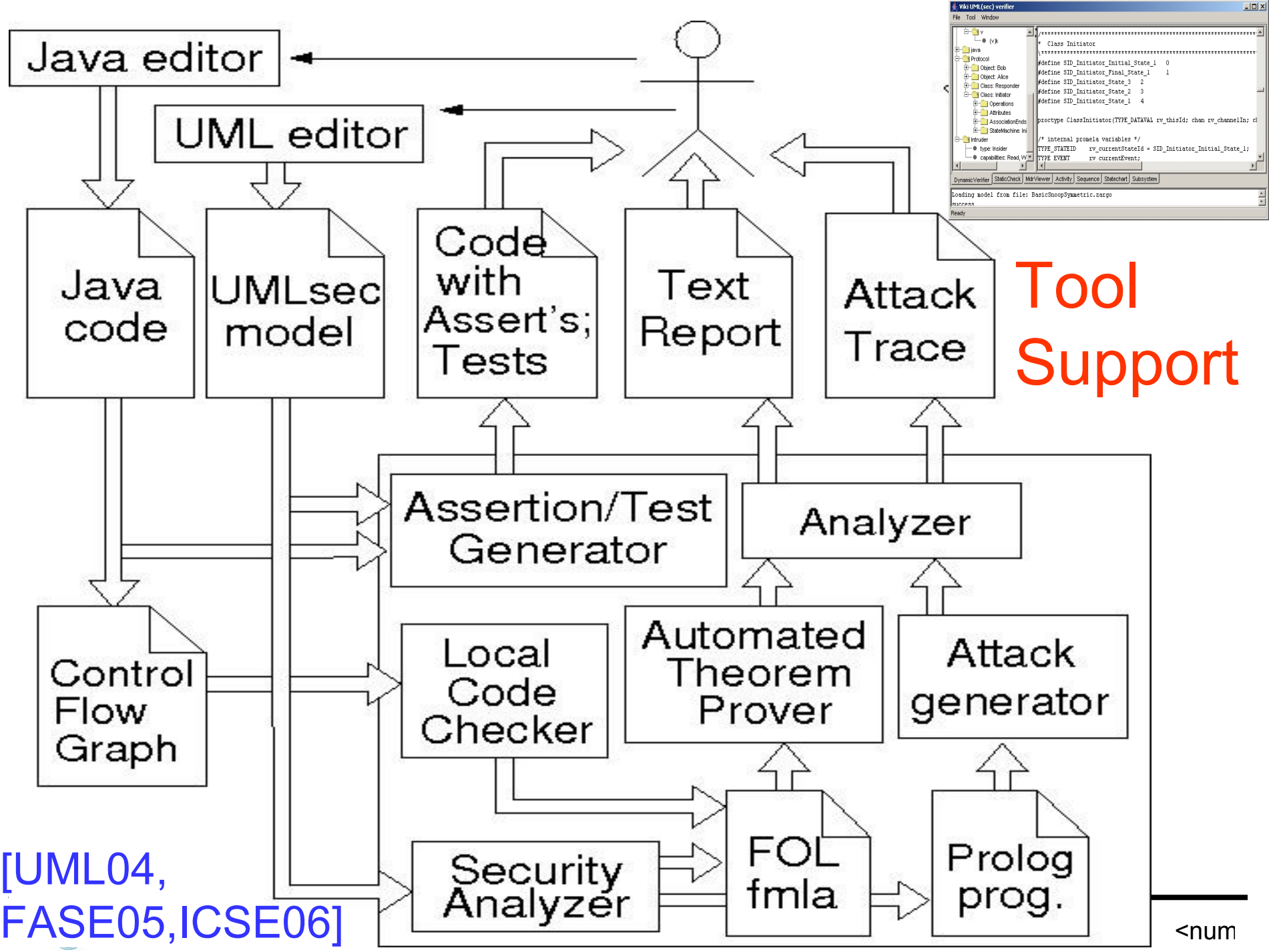
CSDUML Framework: Features

- Framework for analysis plug-ins to access UML models on conceptual level over various UI's.
- Exposes a set of commands. Has internal state (preserved between command calls).
- Framework and analysis tools accessible and available at http://www.umlsec.org .
- Upload UML model (as .xmi file) on website. Analyse model for included critical requirements. Download report and UML model with highlighted weaknesses.

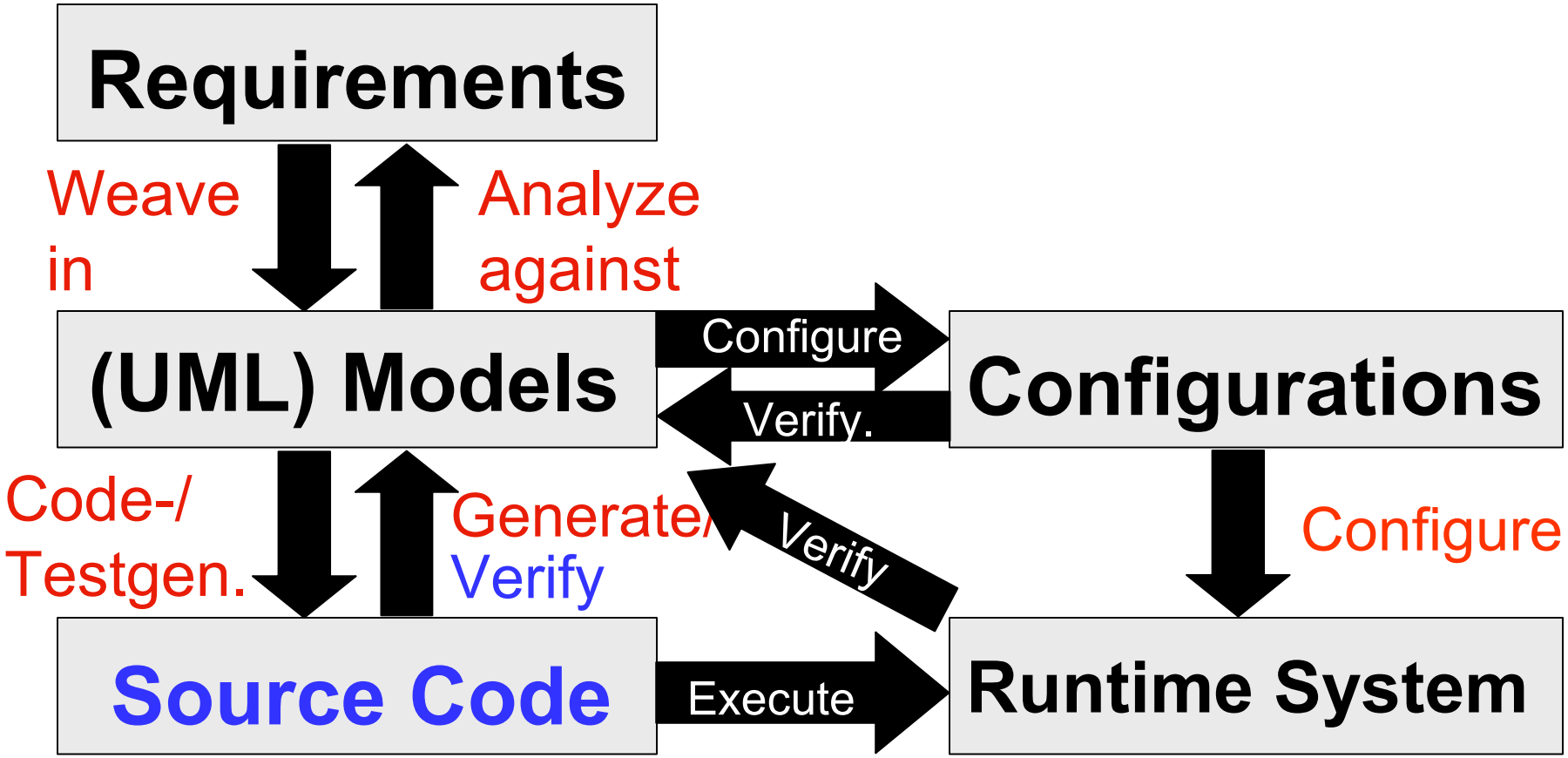
Tool Support

For example:

- consistency checks
- mechanical analysis of complicated requirements on model level (bindings to model-checkers, constraint solvers, automated theorem provers, …)
- code generation
- test-sequence generation
- configuration data analysis against UML.


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]

Roadmap


Requirements
Weave in
Analyze against
(UML) Models
Configure
Verify.
Configurations
Code-/ Testgen.
Generate/ Verify
Verify
Configure
Source Code
Execute
Runtime System

Security Analysis: Model or Code ?

Model:

+ earlier (less expensive to fix flaws)
+ more abstract ➔ more efficient
- more abstract ➔ may miss attacks
- programmers may introduce security flaws
- even code generators, if not formally verified

Code:

+ „the real thing“ (which is executed)

➔ Do both where feasible !

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
- Haneberg, Schellhorn, Grandy, Reif (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.

Model vs. Implementation

[with David Kirscheneder]

„meaning“

„meaning“

compare meaning!

Backtrace assignments

Defined during model creation

Sent and received data

Elements of connections

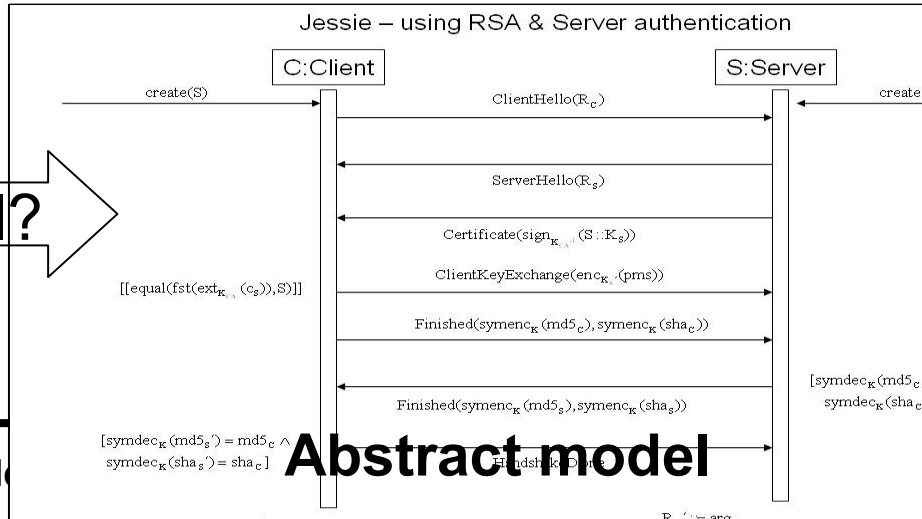
Find

Has

Implement -ation

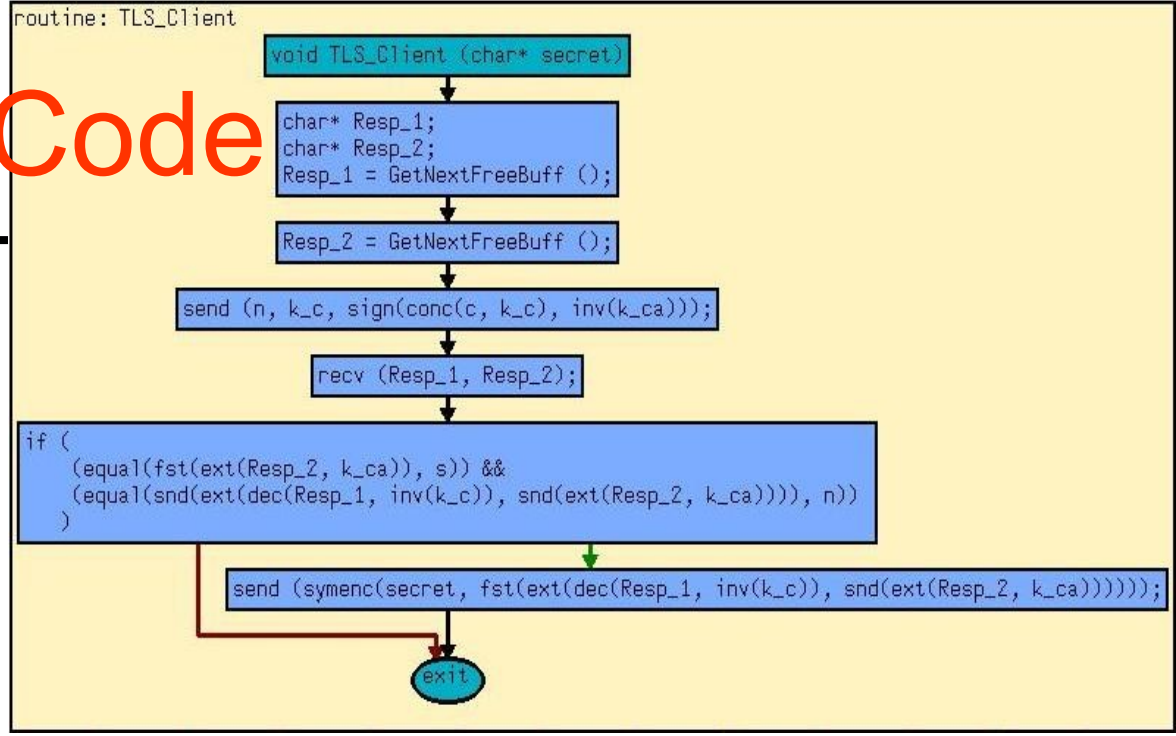
.java

Equal?



Abstract model

Models from Code



Generate control flow graph (e.g. aicall (Absint)).

Transform to state machine:

trans(state,inpattern,condition,action,nextstate)

where action can be outpattern or localvar:=value.

[ASE05,ASE06]

Real Life Challenges

...

ens, O

Experiences

Can generate behavioral models from code (e.g. CFGs). Problem: too concrete

➔ understanding + automated verification hard (even with annotations).

Constructing abstract specifications from practical software is manually intensive.

Code Analysis vs. Model Analysis

Options:

- generate code from models
  ➔ currently not possible in general
- generate models from code
  ➔ challenging
- create models and code manually and verify code against models
  ➔ next slides

Verify Code against Models

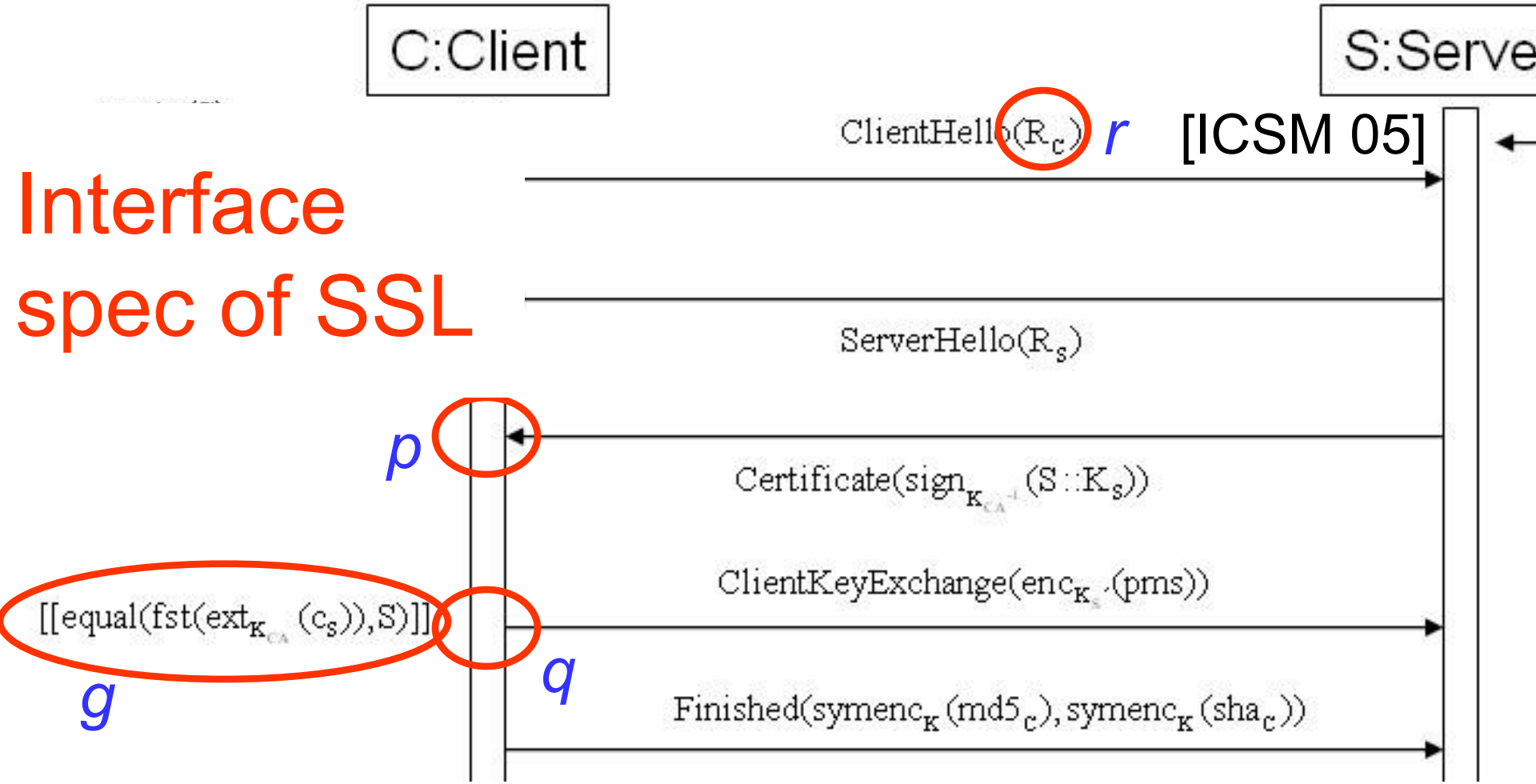
Assumption: Have textual specification.
Then:

- construct interface spec from textual spec
- analyze interface spec for security
- verify that software satisfies interface spec (using run-time verification)

JSSE / Jessie

- Java Secure Sockets Extension (JSSE) contains implementation of SSL.
- Open-source clean-room reimplementation Jessie.
- Applied our approach to fragment of Jessie (SSL handshake using RSA, verifying secrecy of exchanged secret).
- Currently extending the work to JSSE recently made open-source by Sun.

Interface spec of SSL



I) Identify program points:

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

II) Check guards enforced

Parameter der kryptographischen ClientHello Nachricht	Effektiv übertragene Daten der ClientHello Nachricht der Jessie Implementierung
C	type.getValue()
Pver	major
	minor
	((gmtUnixTime >>>
	((gmtUnixTime >>> 16) & 0xFF)
	((gmtUnixTime >>> 8) & 0xFF)
	(gmtUnixTime & 0xFF)
r_c	randomBytes
	sessionId.length
Sid	sessionId
	((suites.size() << 1) >>> 8 & 0xFF)
	((suites.size() << 1) & 0xFF)
LCip	suites_1
	...
	suites_N
	comp.size()
LKomp	comp_1
	...
	comp_N

Implementation (Jessie): Identify Values

Currently do this manually using code assertions

public void write(OutputStream out) throws IOException
{ ... **out.write(randomBytes);** … }

Identify: randomBytes

(in message *ClientHello*)

2nd parameter of Random constructor called by ClientHello.write()

public void write(OutputStream out) throws IOException
{ ... **random.write(out);** ... }

2nd parameter of ClientHello constructor

ClientHello(… , Random random,)
{ ... **this.random = random;** ... }

via Handshake.write()
initialized in SSLSocket.doClientHandshake()

ClientHello clientHello = new ClientHello(...,**clientRandom**,...);

initialization of the used Random object

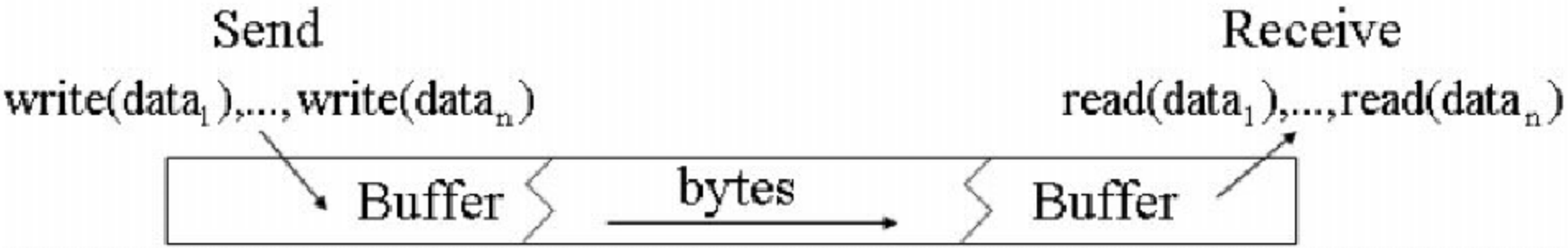
Random clientRandom =
new Random(...,**session.random.generateSeed(28)**);

„meaning“

class SecureRandom (specified in: FIPS 140-2,RFC 1750) of package java.security
Function: generateSeed

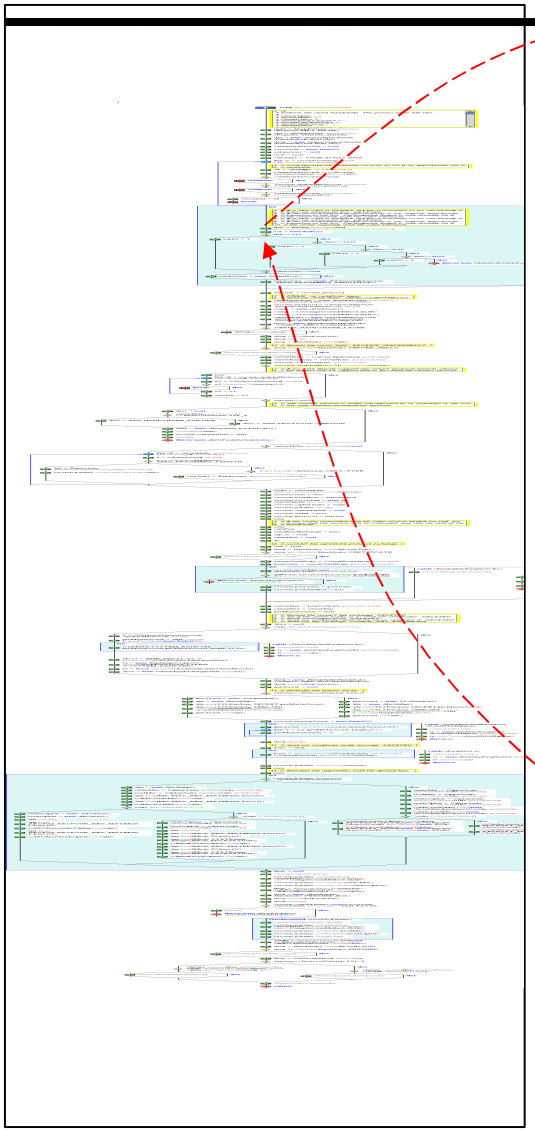
Input / Output

- To extract input/output labels for state machine transitions, analyze input / output mechanism used in the implementation.
- Many implementations (e.g. Jessie and JSSE) use buffered communication where the message objects implement read and write methods. Translate these method calls to input / output labels (need to track successive subcalls).

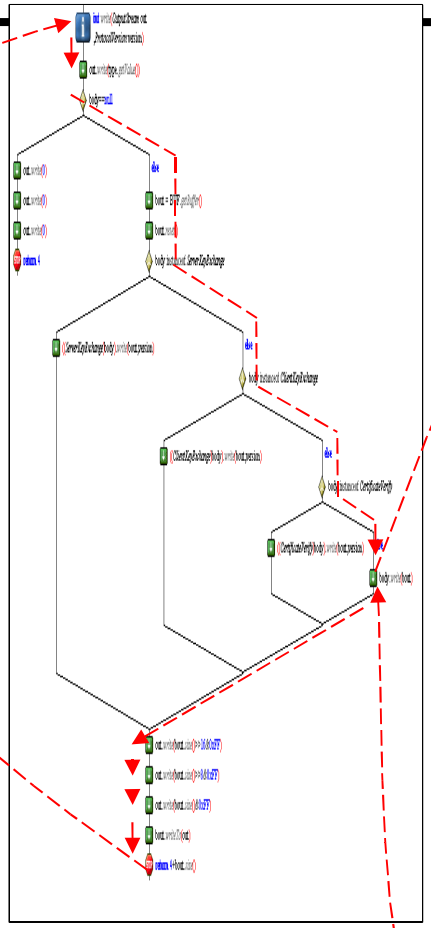


Sending Messages

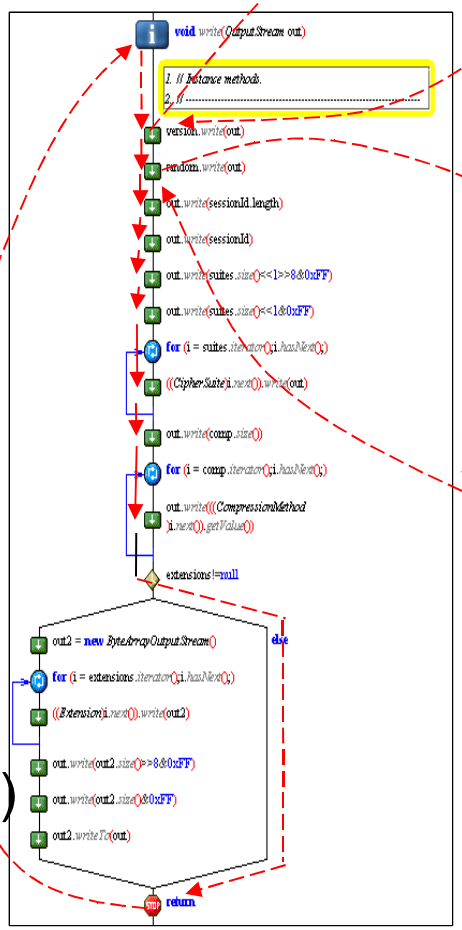
Automate this using patterns



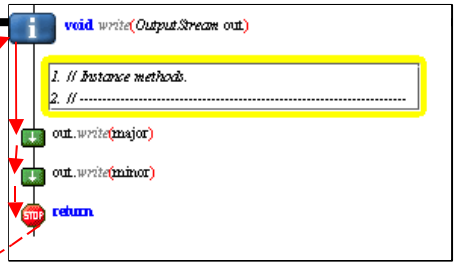
SSLSocket.doClientHandshake()



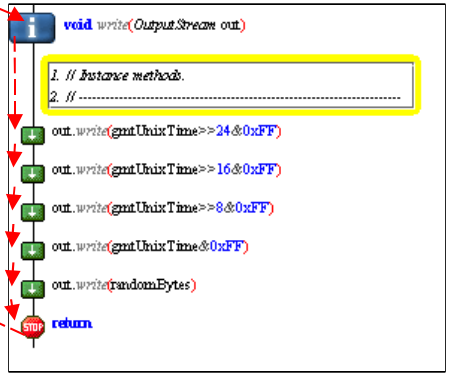
Handshake.write()



ClientHello.write()



ProtocolVersion.write()

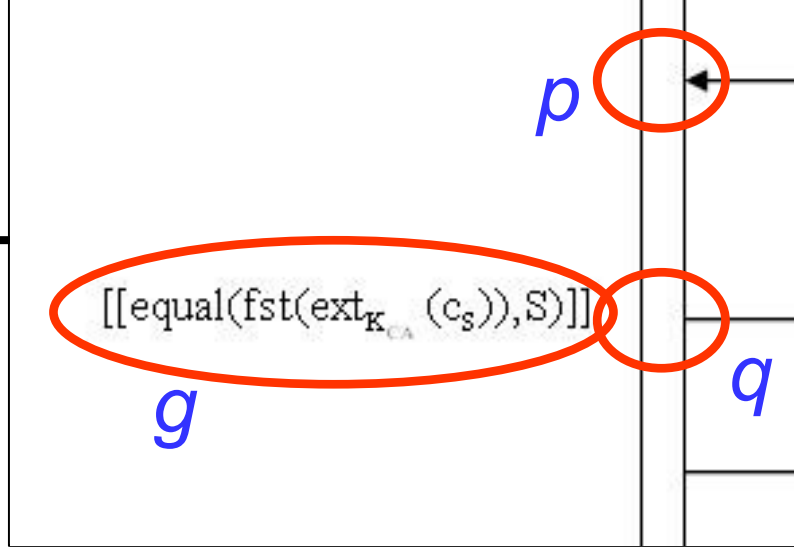


Random.write()

traverse CFG

call of OutputStream.write()

Checking Guards



Guard *g* enforced by code?

b) Generate runtime check for *g* at *q* from diagram: simple + effective, but performance penalty.

c) Testing against checks (symbolic crypto for inequalities). [ICFEM02]

d) Automated formal local verification: conditionals between *p* and *q* logically imply *g* (using ATP for FOL). [ASE06]

```
public void checkServerTrusted(X509Certificate[] chain, String authType)
  throws CertificateException  {…      checkTrusted(chain, authType);    }
```

calls checkTrusted()

Guard:
checkServerTrusted()

```
private void checkTrusted(X509Certificate[] chain,
String authType) throws CertificateException
   {      ...    }
```

calls verify() for every member of certificate chain

```
public void verify(PublicKey key, String provider)
 throws CertificateException, ...
   {      ...    }
```

calls doVerify()

„meaning“

java.security.Signature

- **Initializatize**
- **Update**
- **Verify**

„verifies the signature“

```
private void doVerify(Signature sig,PublicKey key)
   throws CertificateException, ...
  {   ...    sig.initVerify(key);
   sig.update(tbsCertBytes);
   if (!sig.verify(signature))
  {… throw new CertificateException
       ("signature not validated"); …   }  }
```

Execution Path Diagram for method SSLSocket.doClientHandshake()

msg = Handshake.read(din, certType);

p

catch

try

Navigator

only possible way without throwing exception

g

session.trustManager.checkServerTrusted (peerCerts,suite.getAuthType());

serverKex = se

q

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

p

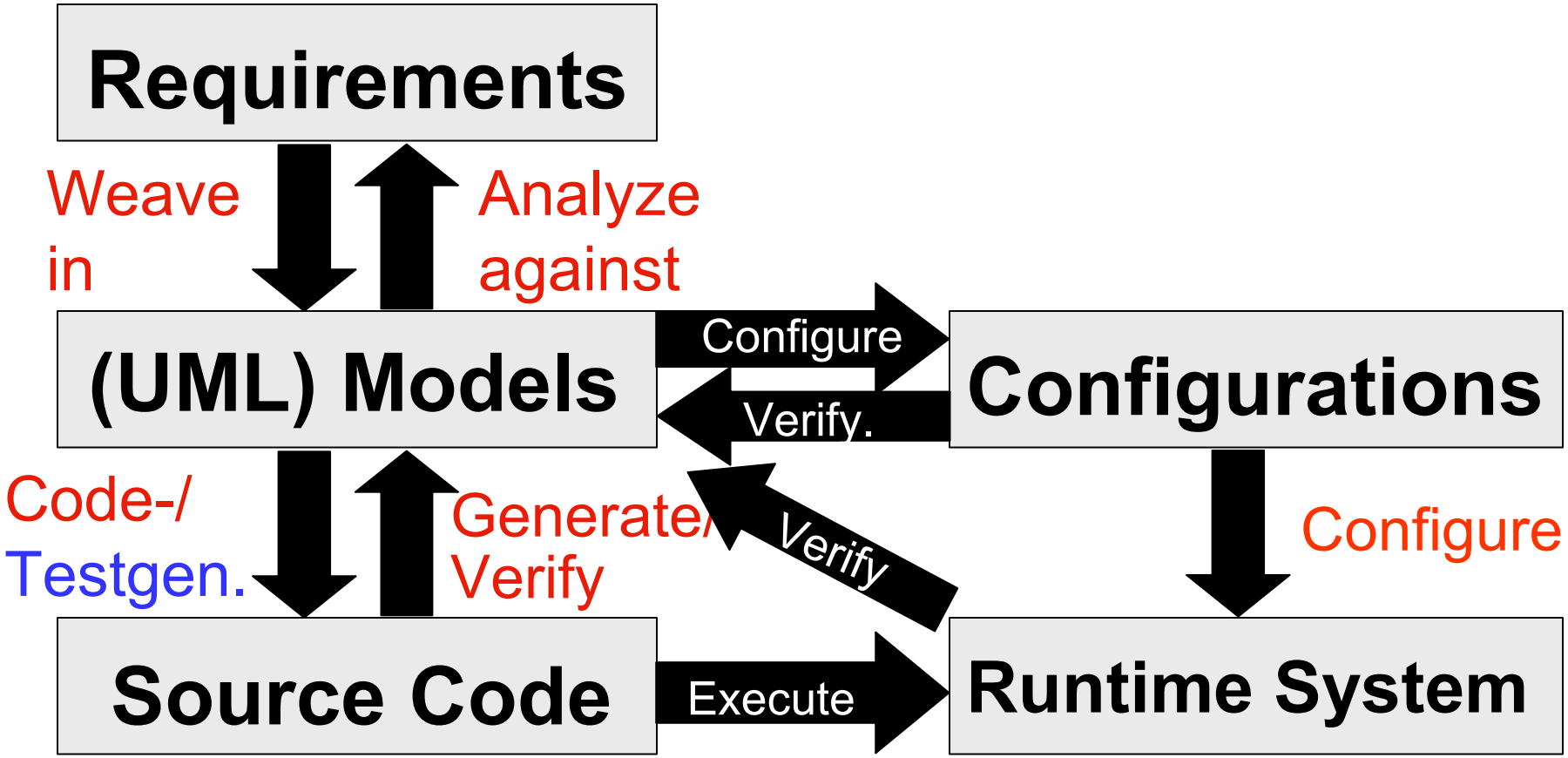
[[equal(fst(ext$_{K_{CA}}$ (c$_S$)),S)]]

q

g

Class Sequence Flow

Roadmap


Requirements
Weave in
Analyze against
(UML) Models
Configure
Verify.
Configurations
Code-/ Testgen.
Generate/ Verify
Verify
Configure
Source Code
Execute
Runtime System

Model-based Testing

Advantages over classical testing:

- Precise measures for completeness.
- Can be formally validated.

Two complementary strategies:

- Conformance testing
- Testing for criticality requirements

Conformance Testing

Classical approach in model-based test-generation (much literature).

Can be superfluous when using code-generation [except to check your code-generator, but only once and for all].

Works independently of real-time requirements.

Conformance Testing: Caveats

- Complete test-coverage still infeasible (although can measure coverage).
- Can only test code against what is contained in model. Usually, model more abstract than code. May lead to „blind spots“.

For both reasons, may miss critical test-cases. Want: „criticality testing“.

Criticality Testing: Strategies

Internal: Ensure test-case selection from models does not miss critical cases: Select according to information on criticality.

External: Test code against possible environment interaction generated from parts of the model (e.g. deployment diagram with information on physical environment).

Criticality Testing

Shortcoming of classical model-based test-generation (conformance testing) motivates „criticality testing“.

Goal: model-based test-generation adequate for critical real-time systems.

Internal Criticality Testing

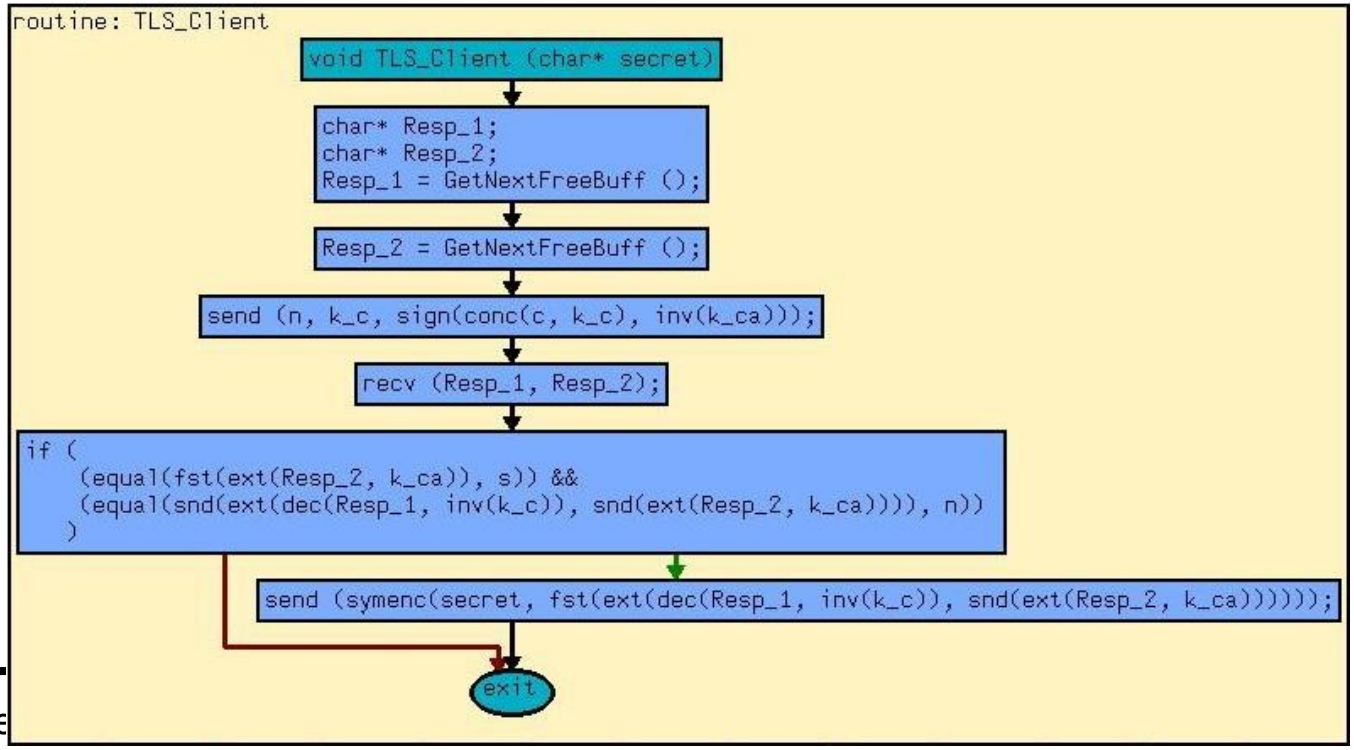
- Need behavioral semantics of used specification language (precise enough to be understood by a tool).
- Here: semantics for simplified fragment of UML in „pseudo-code“ (ASMs).
- Select test-cases according to criticality annotations in the class diagrams.
- Test-cases: critical selections of intended behavior of the system.

External Criticality Testing

Generate test-sequences representing the environment behaviour from the criticality information in the deployment diagrams.

Automated White-Box Testing

- Generate control flow graph.
- Analyze for criticality requirements.
- Use to generate critical test-cases.



Model-based Testing with UML

Meaning of diagrams stated informally in (OMG 2003).

Ambiguities problem for

- tool support
- establishing behavioral properties (safety, security)

Need precise semantics for used part of UML, especially to ensure security requirements.

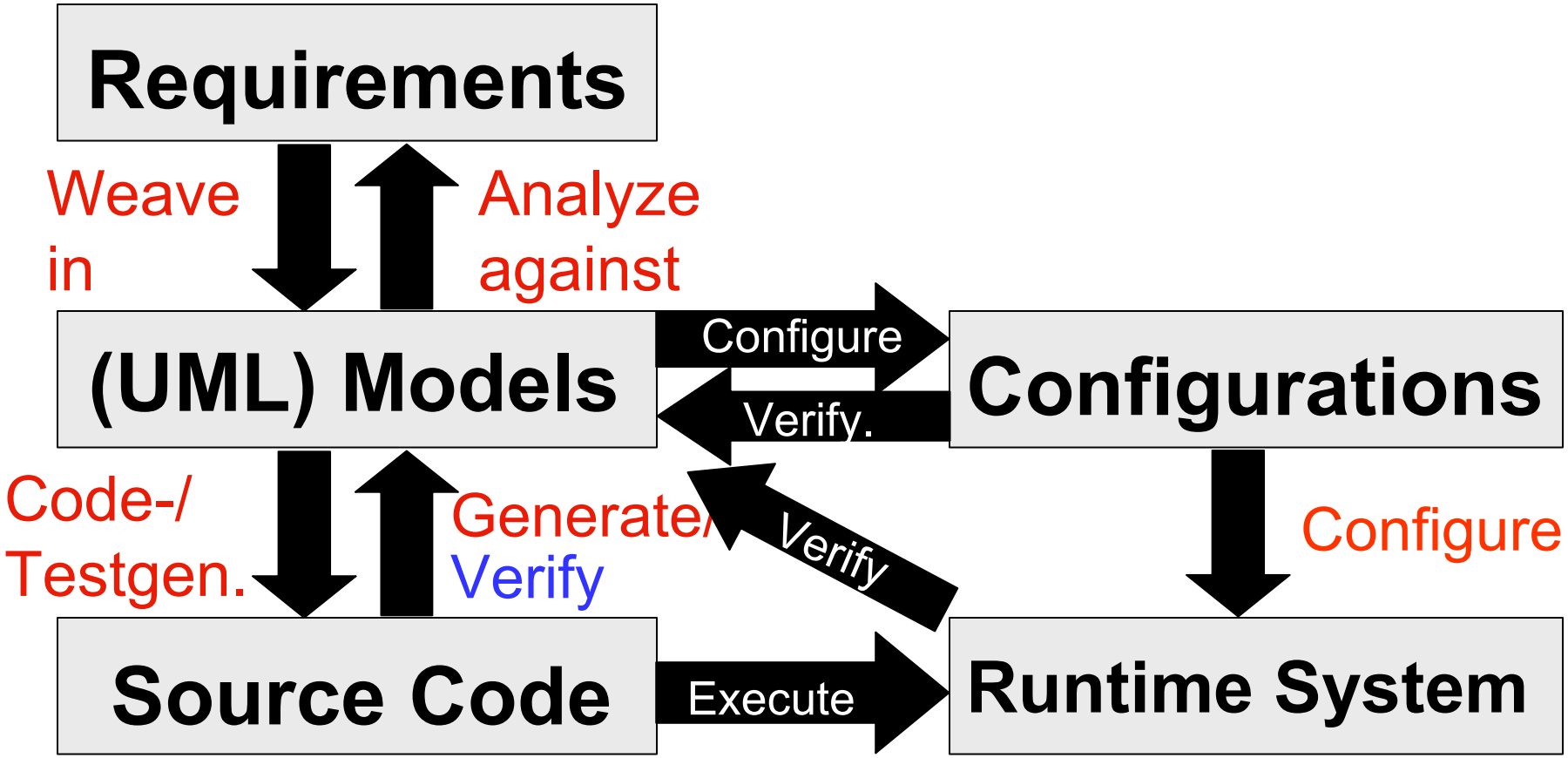
Vulnerability in SSL implementation

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.

[ICSMM07, with Yijun Yu, J. Mylopoulos]

Roadmap


Requirements
Weave in
Analyze against
(UML) Models
Configure
Verify.
Configurations
Code-/ Testgen.
Generate/ Verify
Verify
Configure
Source Code
Execute
Runtime System

Verification of Guards in Code

send: represents send command

g: FOL formula with symbols msg_n representing n^{th} argument of message received before program fragment p is executed

$[d]\ p \models g$: g checked in any execution of p initially satisfying d before any send

write $p \models g$ for $[true]\ p \models g$.

$$\frac{}{[d]\ \text{if } c \text{ then } p \text{ else } q \models g}\ (c \wedge d \Rightarrow g,\ \text{no send in } q)$$

Loops

- In automated verification, often only consider finite number of iterations.
- Here: in translation to logic, replace variables in loops by infinite arrays (index: loop counter).
- Note: using ATP, don‘t need to worry about finding loop invariants.
- General problem undecidable, but at our level of abstraction for crypto-protocols not a problem since emphasis on interaction rather than computation.

Loops: Example

Example:

```
while (true)
{ k = a + 1;
  a = b + k;
  b = b + 1; }
```

SSA:

```
while (true)
{ k = a0 + 1;
  a1 = b0 + k;
  b1 = b0 + 1; }
```

TPTP:

```
input_formula(ForLoop_axiom_ID1,axiom,(
![I]: (equal (k[I], sum(a0[I],1)) &
       equal (a1[I], sum(b0[I],k[I])) &
       equal (b1[I], sum(b0[I],1)) &
       equal (a0[succ(I)],a1[I]) &
       equal (b0[succ(I)],b1[I]))))).
```

Concurrent threads

Identify maximal transition paths in CFG between points where shared variables written or read.

In translation to logic, consider possible interleavings of threads by defining:

ϕ from predicates PRED(Pi) as above (for each path i)

ψ assigning variables according to given interleaving

Join formulas $\psi \Rightarrow \phi$ together by conjunction.

Abstraction by Code Annotations

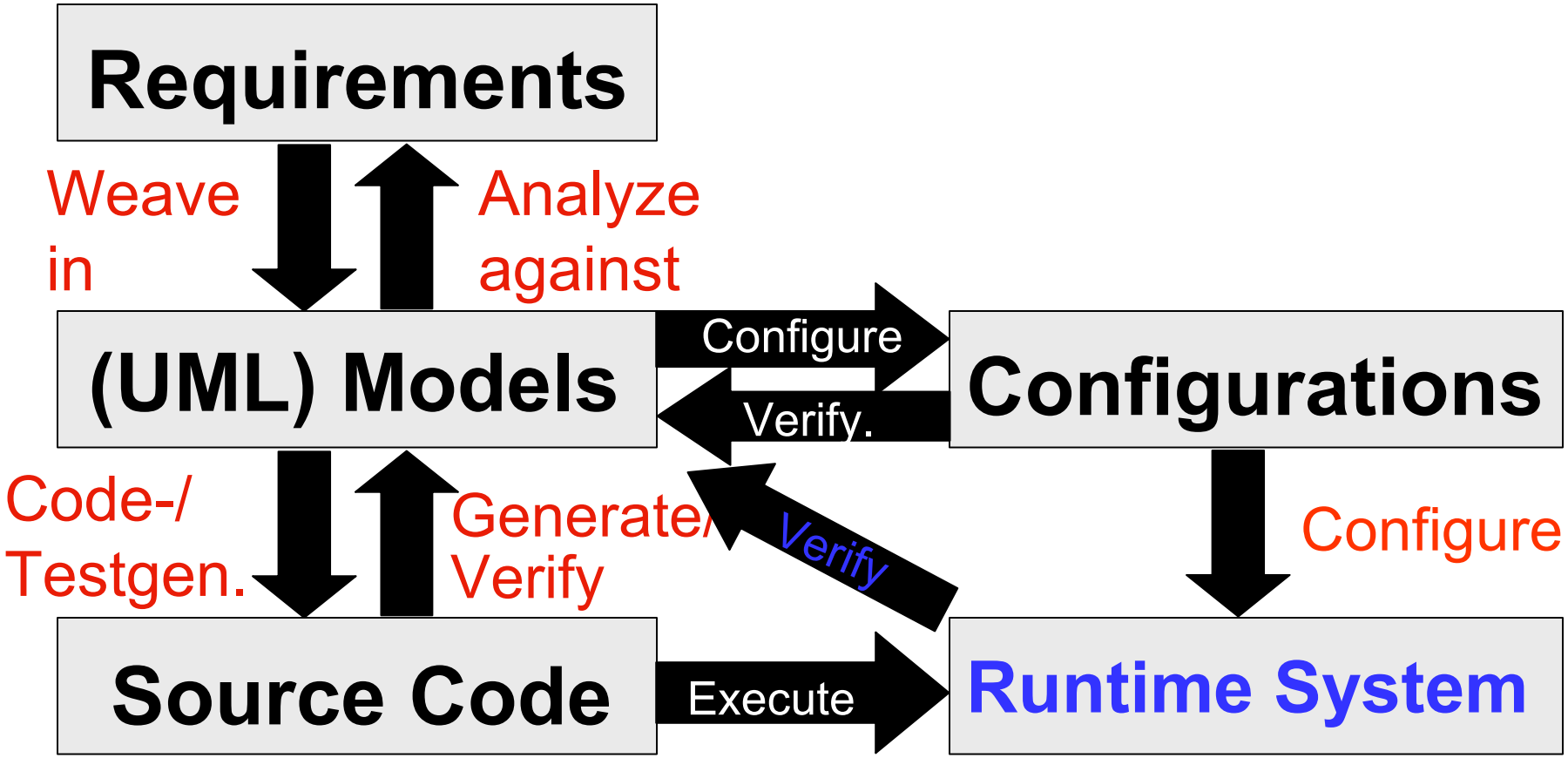
```
//@J2SD_ANN (<<method name>>)
//@J2SD_CONN (<<trigger>>; <<guard>>;
   <<effect>>)
//@J2SD_INSERT (<<value>>)
//@J2SD_AXIOMS (<<value>>)
// <<FOL axioms>>
//@J2SD_AXIOMS_END
```

Similarly for variables / constants.

Modular Verification

- For program fragment p, generate set of statements derive(L,C,E) such that adversary knowledge is contained in every set K that:
  - for every list l of values for the variables in L that satisfy the conditions in C contains the value constructed by instantiating the variables in the expression E with the values from l
- When considering single protocol run, can construct finite set of such statements similar to FOL formulas from security analysis.

Roadmap


Requirements
Weave in
Analyze against
(UML) Models
Configure
Verify.
Configurations
Code-/ Testgen.
Generate/ Verify
Verify
Configure
Source Code
Execute
Runtime System

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 ?

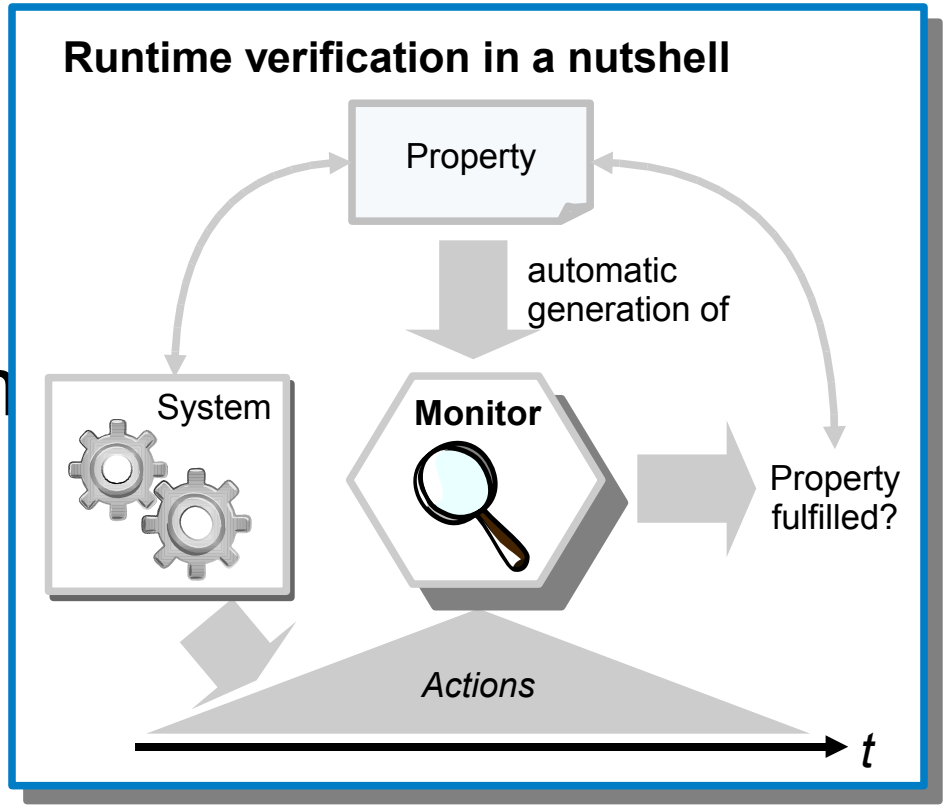
➔ Run-time verification.

Runtime Verification using Monitors

- Dynamic verification technique on the actual system.

[A. Bauer]

- Essentially a symbiosis of model-checking and testing.
- "Lazy model-checking": only check the system traces which are executed, when they are executed.



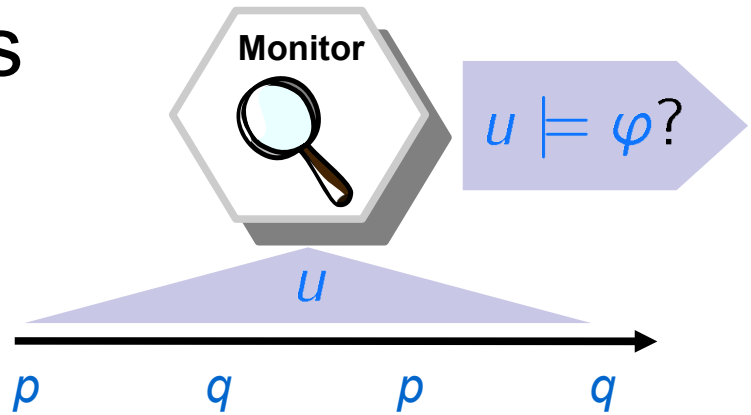
Formal underpinnings

[A. Bauer]

- 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*

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

Semantics

$$
\begin{array}{lcl}
w, i \models \mathit{true} & & \\
w, i \models \neg\varphi & \Leftrightarrow & w, i \not\models \varphi \\
w, i \models p \in AP & \Leftrightarrow & p \in w(i) \\
w, i \models \varphi_1 \vee \varphi_2 & \Leftrightarrow & w, i \models \varphi_1 \vee w, i \models \varphi_2 \\
w, i \models \varphi_1 \boldsymbol{U} \varphi_2 & \Leftrightarrow & \exists k \geq i.\ w, k \models \varphi_2 \wedge \\
 & & \forall i \leq l < k.\ w, l \models \varphi_1 \\
w, i \models \boldsymbol{X}\varphi & \Leftrightarrow & w, i+1 \models \varphi
\end{array}
$$

[A. Bauer]

We write $w \models \varphi$, if and only if $w, 0 \models \varphi$, and use $w(i)$ to denote the ith element in w. (w word, i position)

Write F phi for true U phi ("eventually phi"); G phi for not F not phi ("globally phi"); phi1 W phi2 for G phi1 or (phi1 U phi2) (weak-until)

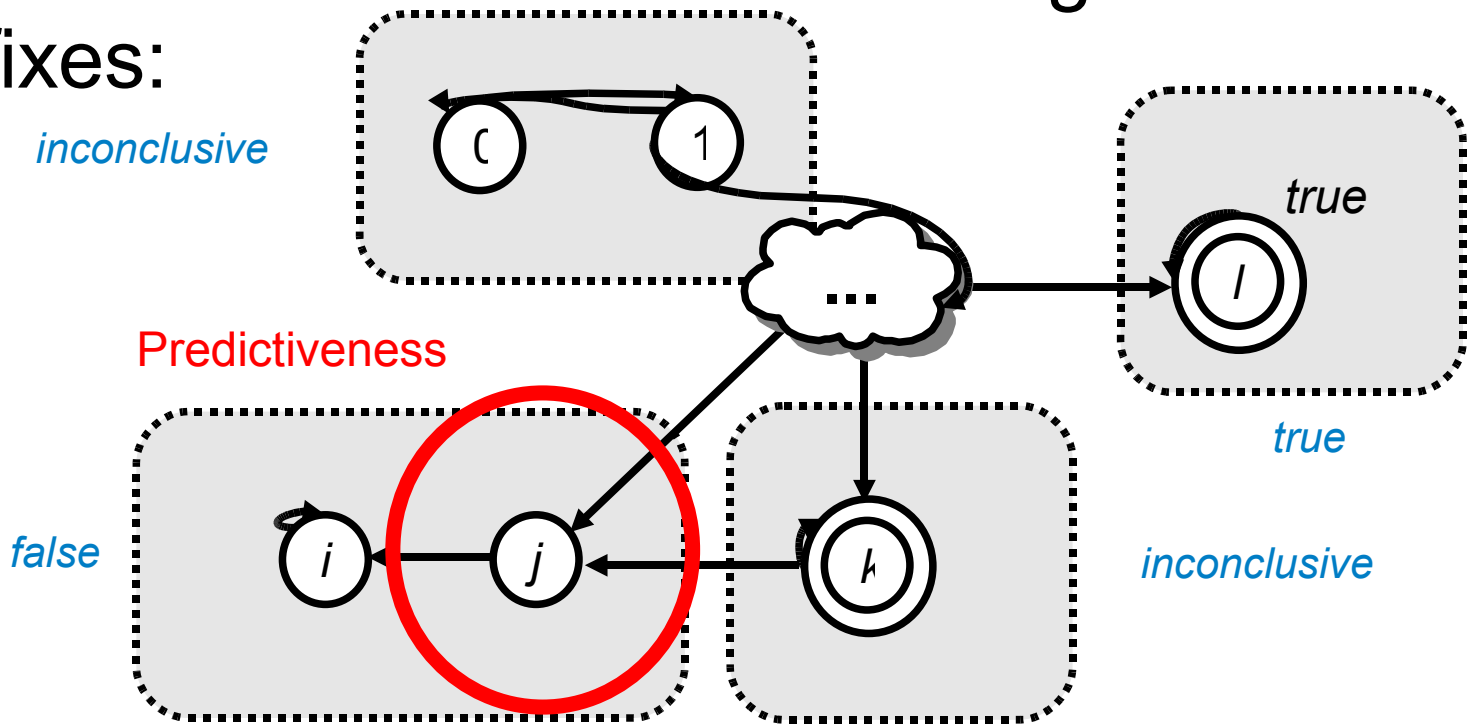
Monitoring-friendly LTL semantics

3-valued semantics:

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

[A. Bauer]

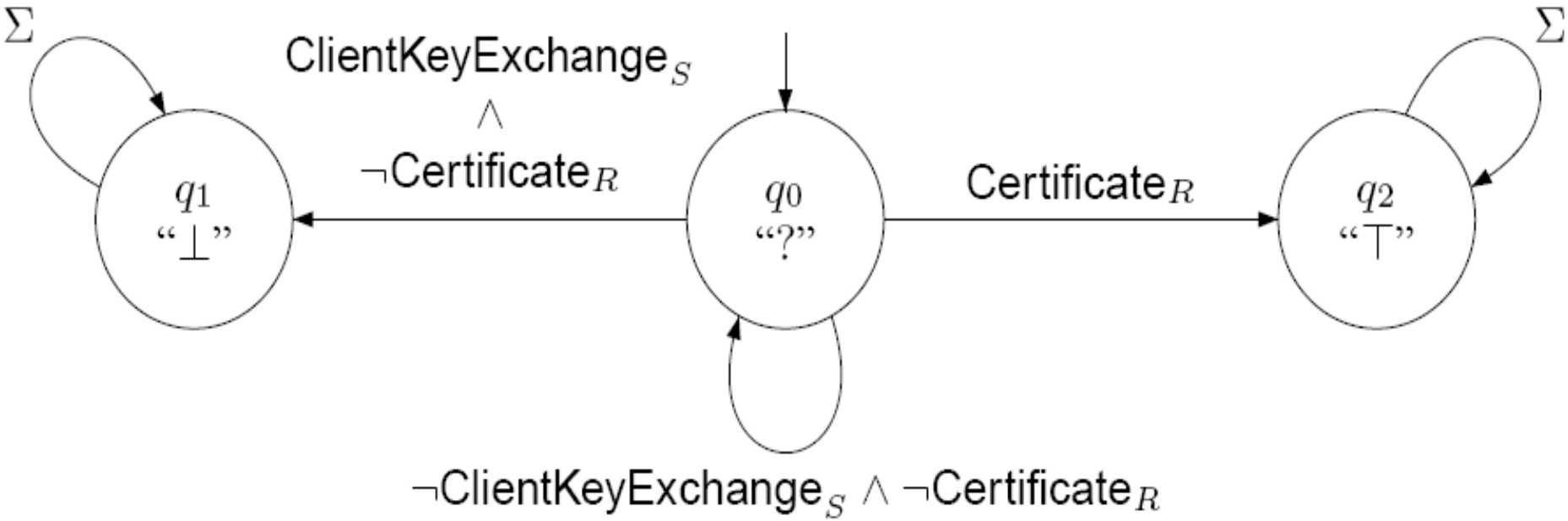
Gives finite-state machines for detecting *minimal* bad prefixes:



ClientKeyExchange

[SESS 08, with A. Bauer]

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$_S$ U Certificate$_R$.

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

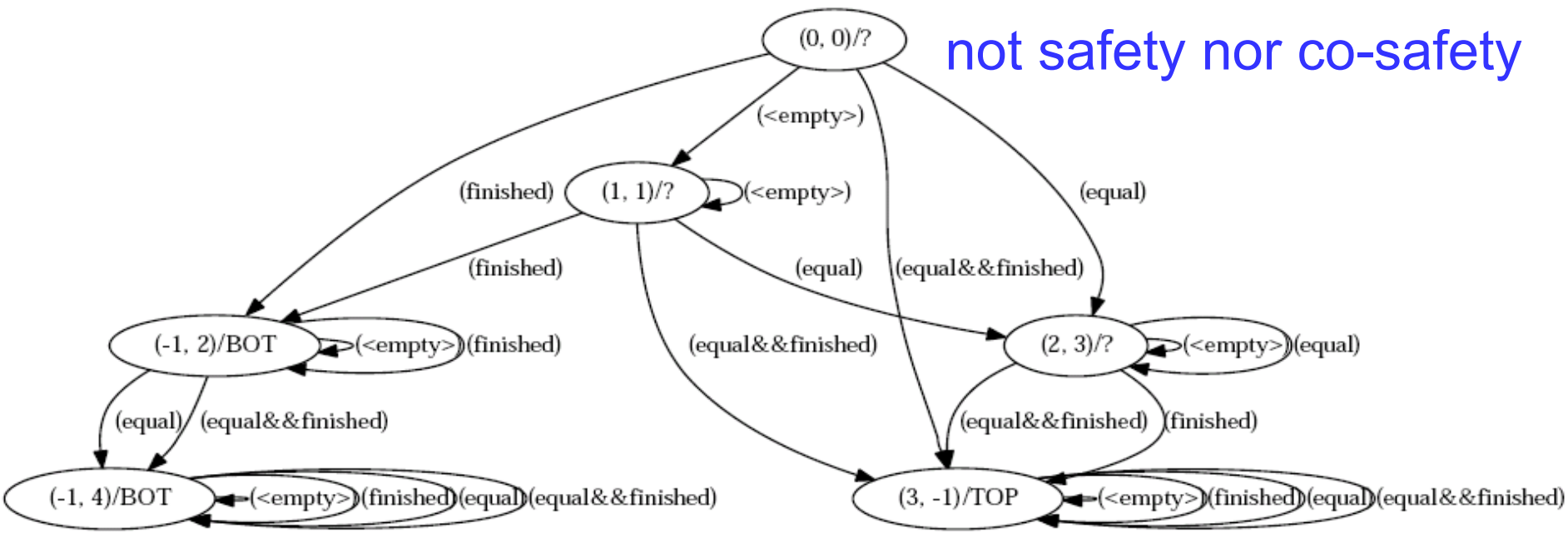
Server Finished

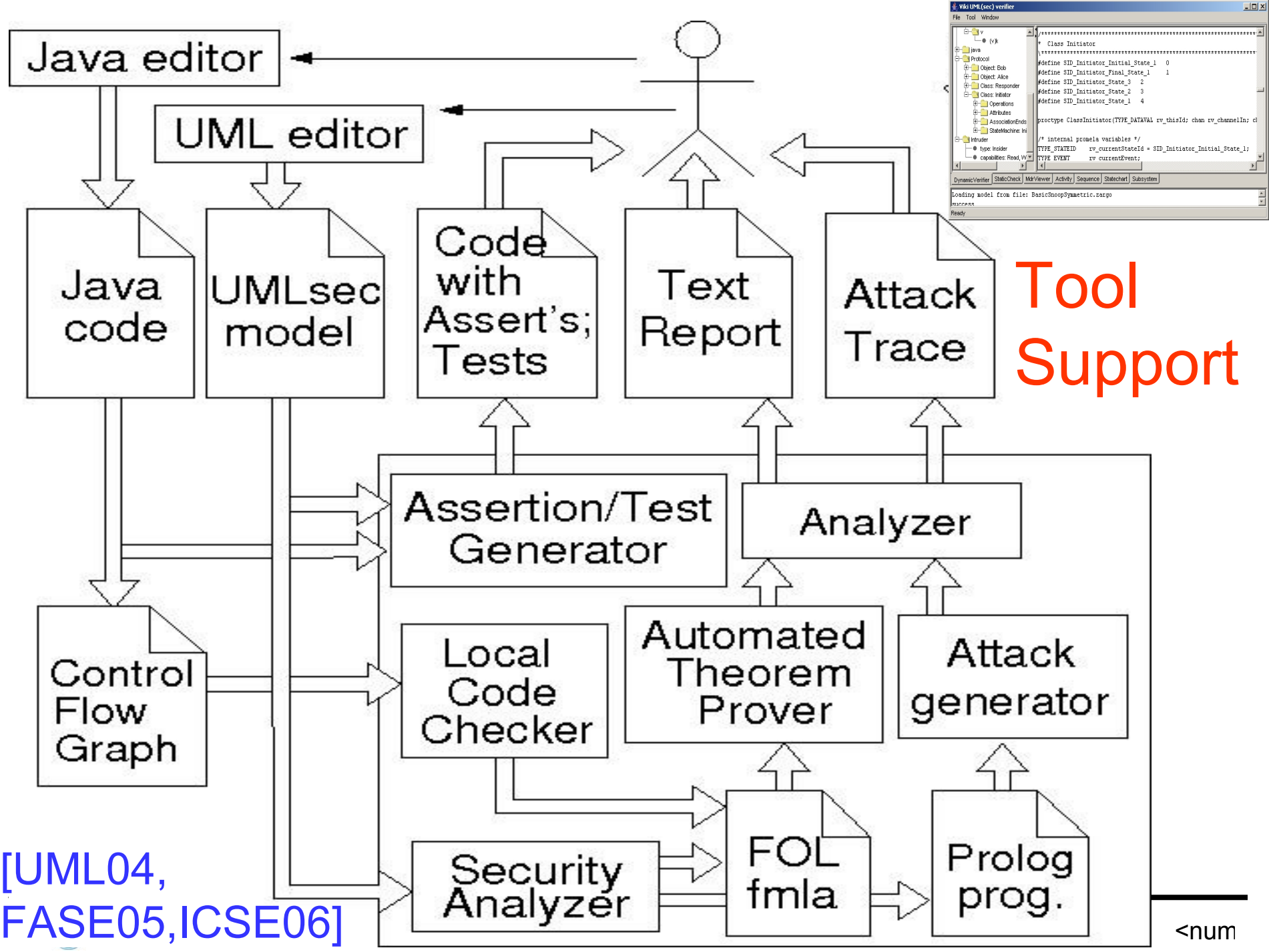
- Server will not send Finished message before MD5 received in Client`s Finished message equal to MD5 created by server. Then sends out eventually.

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

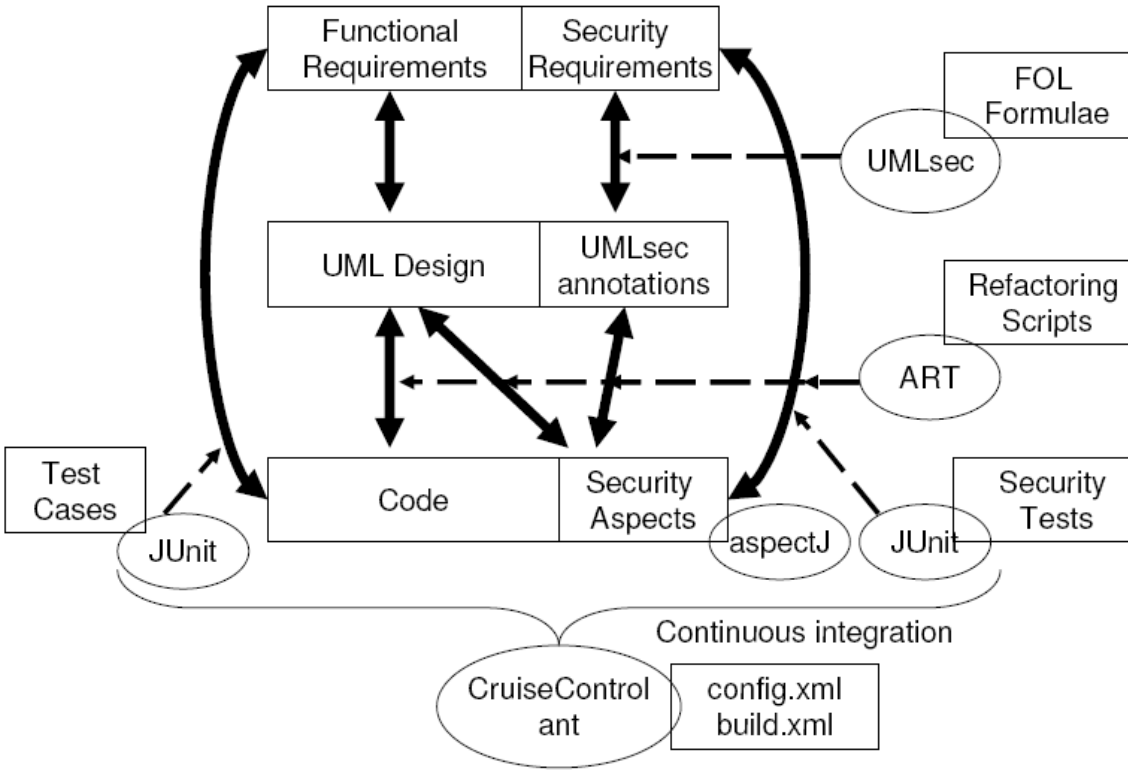



Java editor
UML editor
Java code
UMLsec model
Code with Assert's; Tests
Text Report
Attack Trace
Tool Support
Assertion/Test Generator
Analyzer
Control Flow Graph
Local Code Checker
Automated Theorem Prover
Attack generator
Security Analyzer
FOL fmla
Prolog prog.
[UML04, FASE05,ICSE06]

Tracing Security Requirements

- Tracing security requirements to models… [CAISE 06]
- … reconciling them with other non-functional requirements such as fault-tolerance, performance [UML 04, JSS 07]
- … and from models to code [ASE 07, ICSM 08, ASE 08, w. Y. Yu]
- For legacy systems need to extract security domain knowledge from the code.

  [CSMR 07, CSMR 08, IPCP 08, w. D. Ratiu]



Applications of MBSE

Analyzed designs / implementations / configurations for

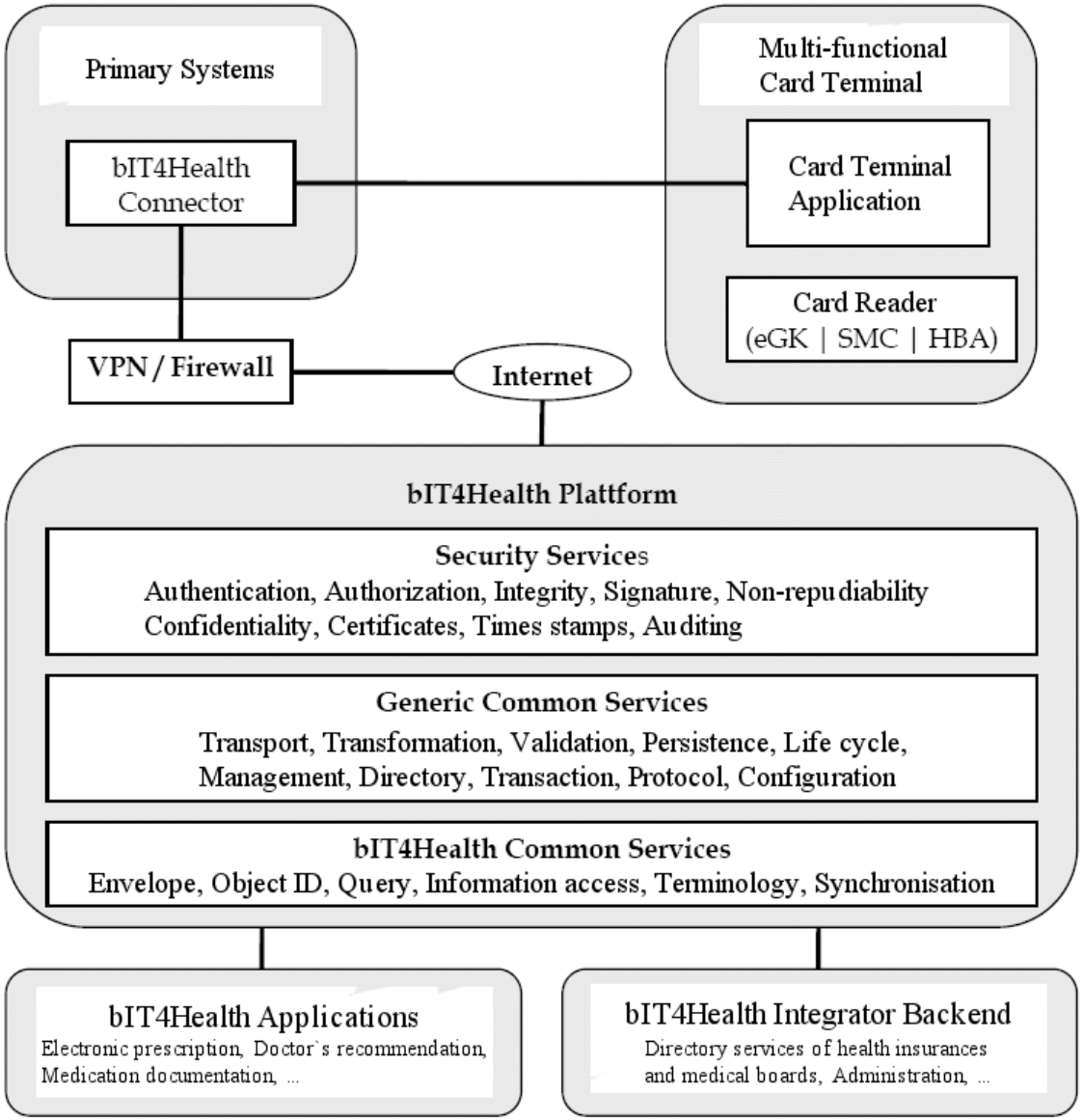
- biometry, smart-card or RFID based identification
- authentication (crypto protocols)
- authorization (user permissions, e.g. SAP systems)

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

German Health Card Architecture

- Analyzed architecture against security requirements using UMLsec
- Detected several security weaknesses in the architecture

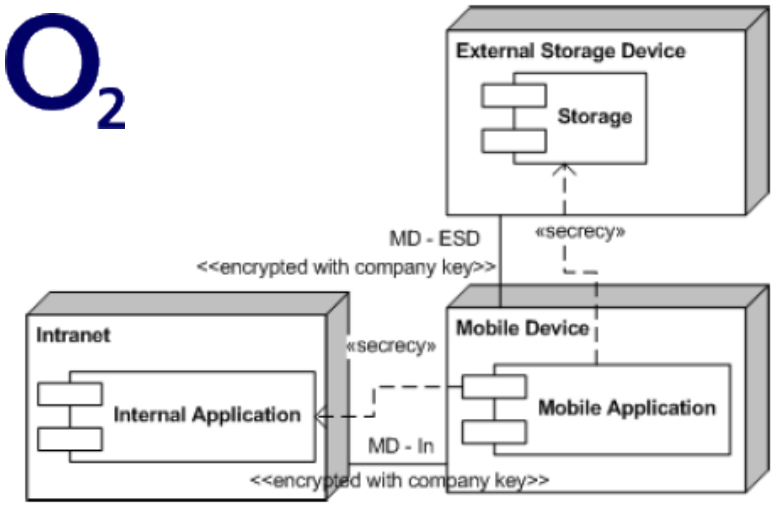
[Meth. Inform. Medicine 08]



Mobile Communications

[ICSE 08]

- Application of Model-based Security Assurance at Mobile Communication Systems at O2 (Germany)
- All 62 relevant security requirements from security policy successfully established using the approach
- Model-based development does incur extra effort.
- Seems manageable when applied to critical system core.
- Justifiable in case of high assurance needs (security).
- Compares favorably with other assurance/same trustworthiness.
- UMLsec well-suited for mobile communication systems.



[ICSE 07]

Intranet Information System

MetaSearch Engine: Personalized search in company intranet (including password protected).

Some documents highly security-critical.

BMW Group

More than 1,000 potential users, index 280,000 documents, allow 20,000 queries per day.

Seamlessly integrated in enterprise-wide security reference architecture. Provides security services to applications, including user authentication, role-based access control, global single-sign-on and hook-up of new security apps.

Successfully analyzed using model-based security.

Bank Application

Security analysis of web-based banking application, to be put to commercial use (clients fill out and sign digital order forms).

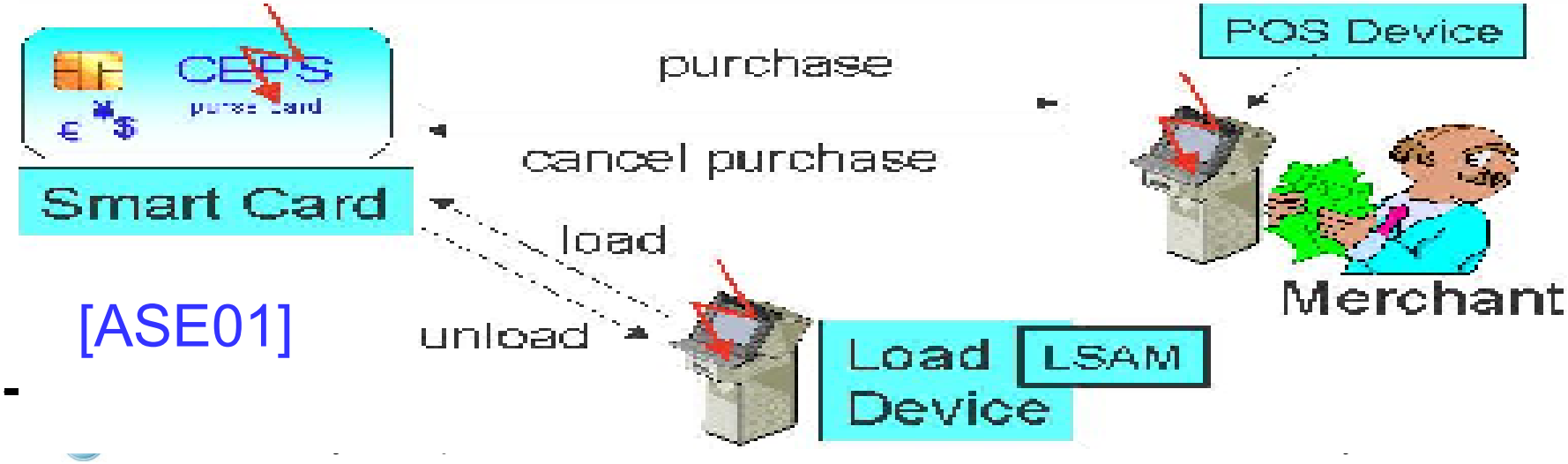
Layered security protocol (first layer: SSL protocol, second layer: client authentication protocol)

Security requirements:

- confidentiality
- authenticity

Common Electronic Purse Specifications

Global elec. purse standard (Visa, 90% market).
Smart card contains account balance, performs crypto operations securing each transaction.
Formal analysis of load and purchase protocols: three significant weaknesses: purchase redirection, fraud bank vs. load device owner.



[ASE01]

Biometric Authentication System

- In development by company in joint project. Uses bio-reference template on smart-card. Analyze given UML spec.
- Discovered three major weaknesses in subsequently improved versions (misuse counter circumvented by dropping / replaying messages, smart-card insufficiently authenticated by mixing sessions). [ACSAC05]
- Here: consider different protocol from public sources but with similar problems.

How does it compare ?

- Empirical study to compare classical vs. model-based testing: embedded software / Automotive (window controller). In cooperation with colleagues from BMW / Elektrobit.

Modelchecking
Examines an abstract model
Cheap and early verification (without setting up complex in-the-loop-test environments)
Proof of correctness of properties possible
Uses selected user specific properties

Testing
Examines a physical or concrete system
In-the-loop-tests take place in an environment near to the real one
No proof of correctness of properties possible
Uses often many, superficial test cases

THIS FRIDAY !

Conclusions

Model-based vs Code-based Verification using UMLsec:

- formally based approach
- automated tool support
- industrially used methods
- integrated approach (source-code, configuration data)

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

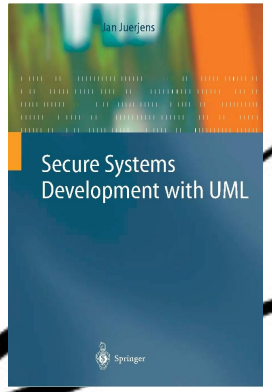
Ongoing Work

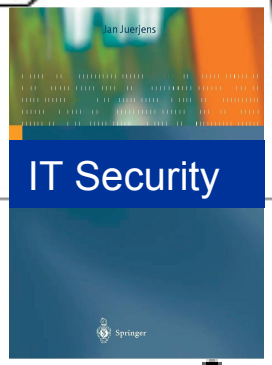
- Security Verification of Crypto Protocol Implementations in C: Use VCC to verify C code. (with Andy Gordon, MSR Cambridge; RS Industrial Fellowship & 2 PhD projects)
- Modelling for Compliance (EPSRC CASE PhD project with British Telecom)
- Security Engineering for Lifelong Evolvable Systems (EU FP7 Integrated Project): HIRING NOW: 2 Postdocs !
- RS Joint International Project with TU Munich on Formal Security Analysis of Cryptoprotocol Implementations
- RS Joint International Project with NII Tokyo Relating Security Requirements and Design

Overview

Security Engineering

- Patterns
- Services
- Architectures
- Components
- Aspects
- Layers
- Requirements
- Processes
- Models
  - UMLsec
  - Formal Semantics
  - Analysis Framework
  - Model Checking
  - Autom. Theorem Proving
  - Refinement
- Applications
  - CEPS
  - Biometry
  - Cryptokey
  - Jessie
  - TLS variant
  - Wiesncard
- Code
  - Runtime Checks
  - Model-based Testing
  - Autom. Theorem Proving
- Foundations





Dependable Systems Development

- Fault-tolerance
- Reliability
- Real-time
- Security
  - Engineering
  - Management
    - Risk Assessment
    - Permissions
    - Business Processes
    - Security Investment
    - Firewall Configurations

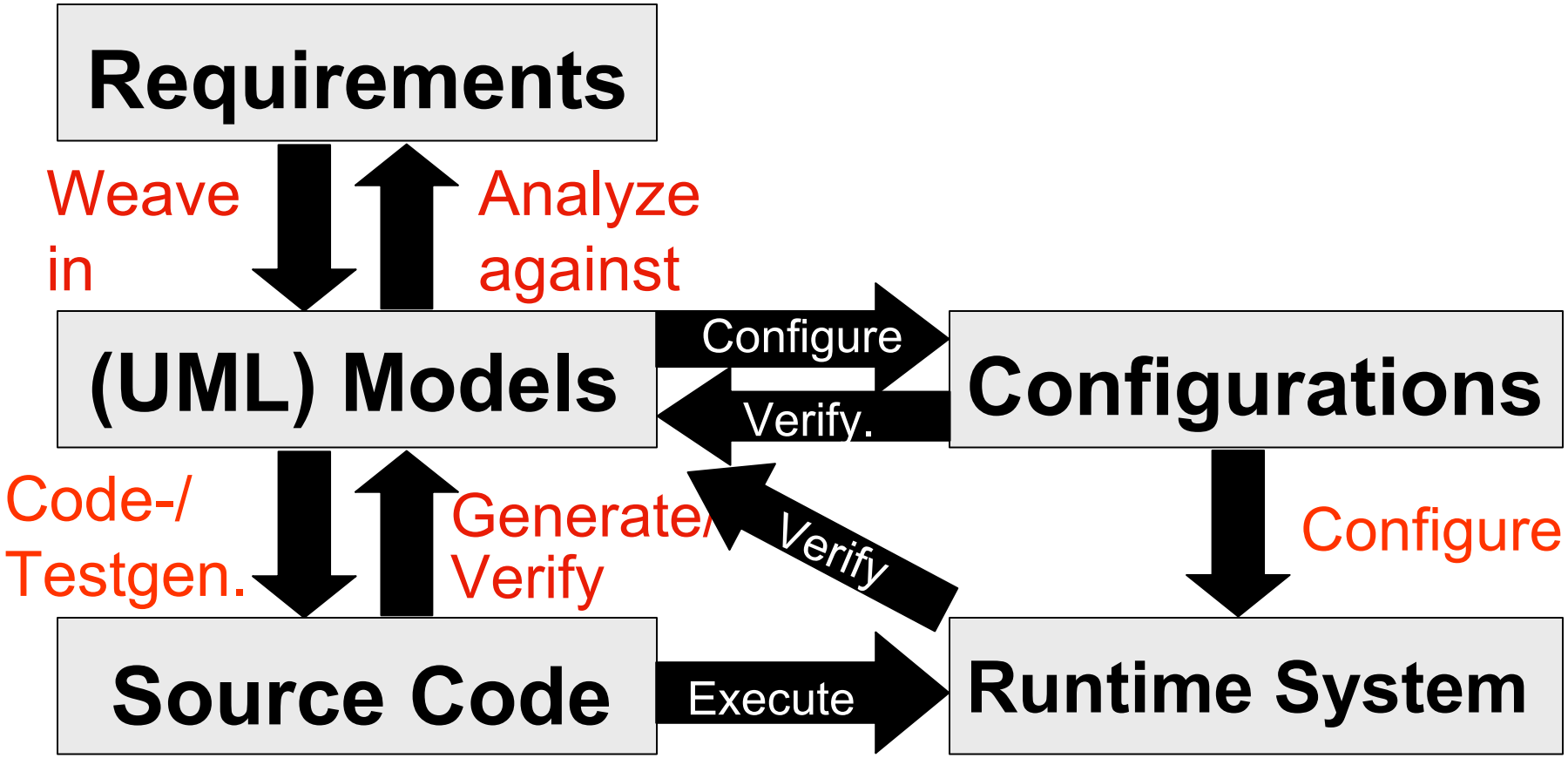
Questions?

More information (papers, slides, tool etc.):

http://www.jurjens.de/jan

J.Jurjens@open.ac.uk

Roadmap


Requirements
Weave in
Analyze against
(UML) Models
Configure
Verify.
Configurations
Code-/ Testgen.
Generate/ Verify
Verify
Configure
Source Code
Execute
Runtime System