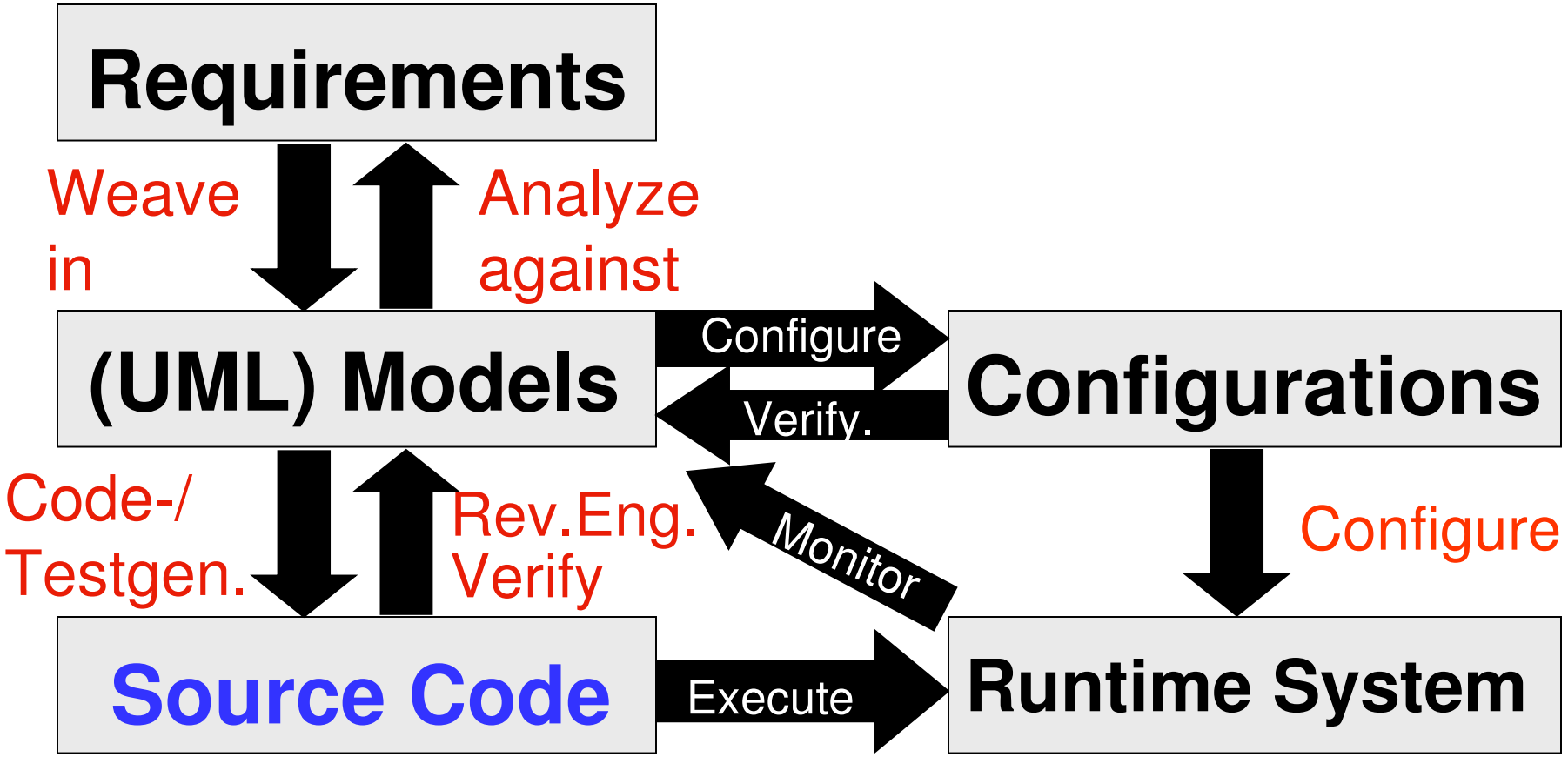
8 Models vs. Code



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
- 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:

- 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

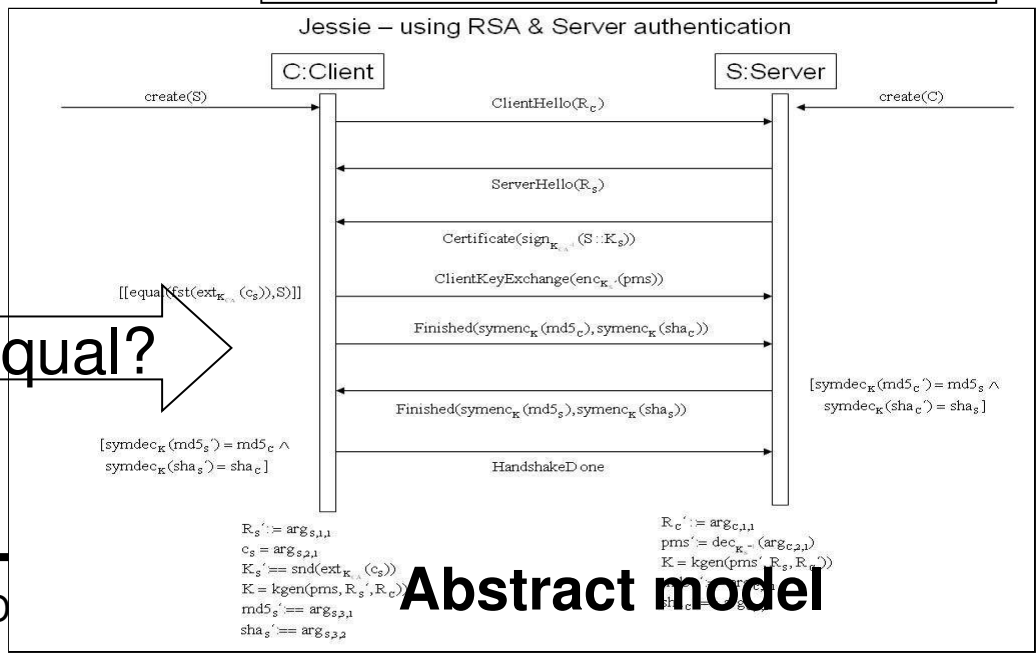
Elements of connections

Find

Implement -ation

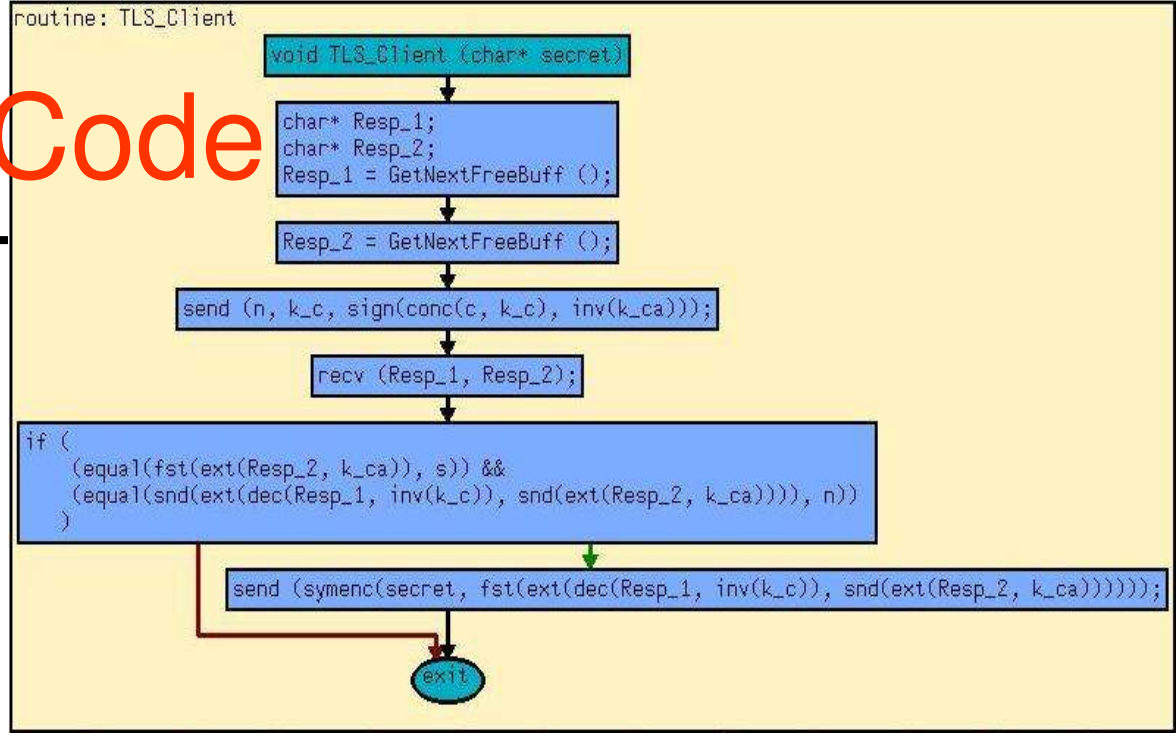
.java

Equal?



Abstract model

Models from Code



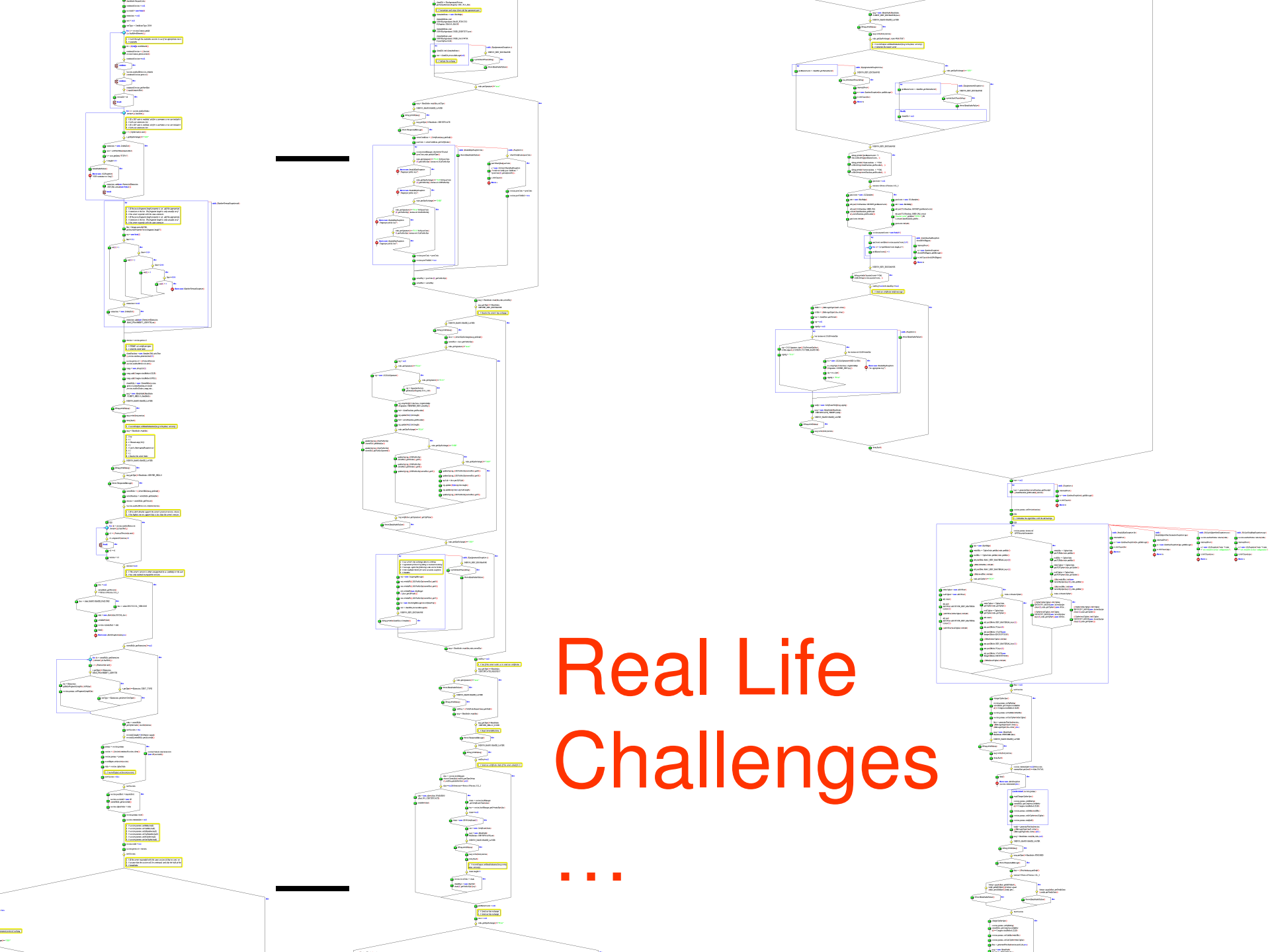
Generate control flow graph (e.g. aicall (Absint)).

Transform to state machine:

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

where action can be outpattern or localvar:=value.

[ASE05,ASE06]


Real Life
Challenges
...

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

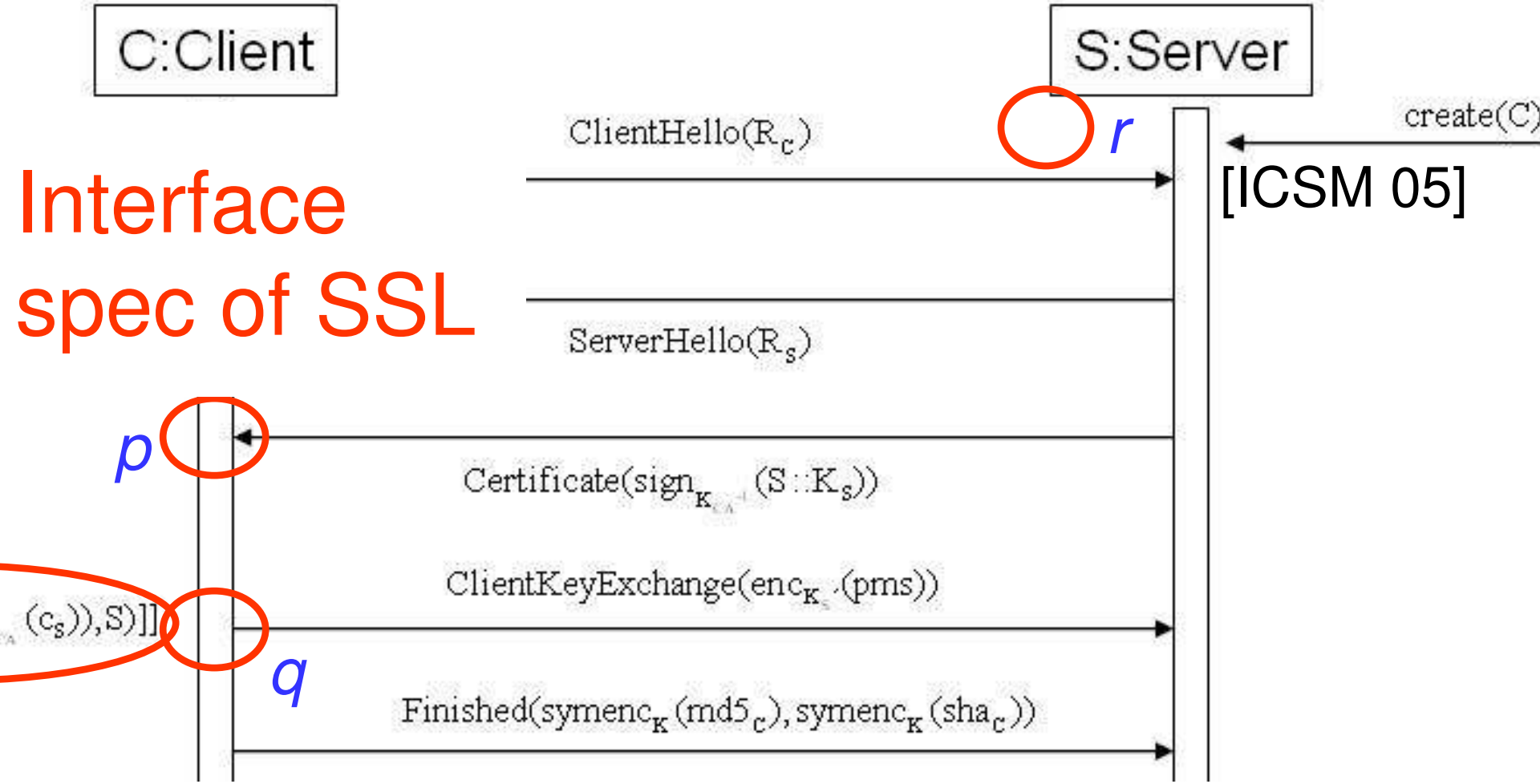
Assumption: Have textual specification. Then:

- 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



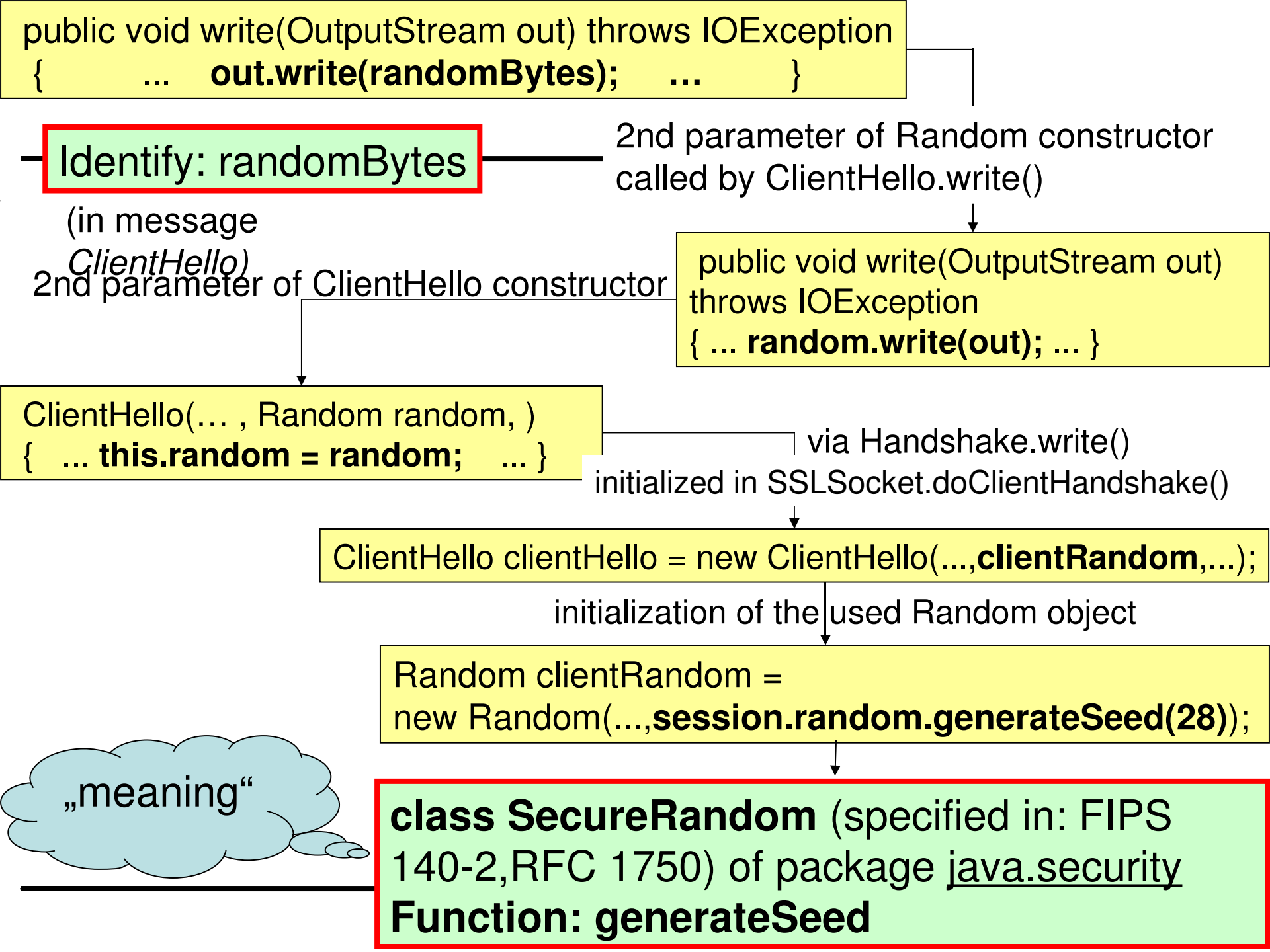
- Identify program points:
  value (*r*), receive (*p*), guard (*g*), send (*q*)

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

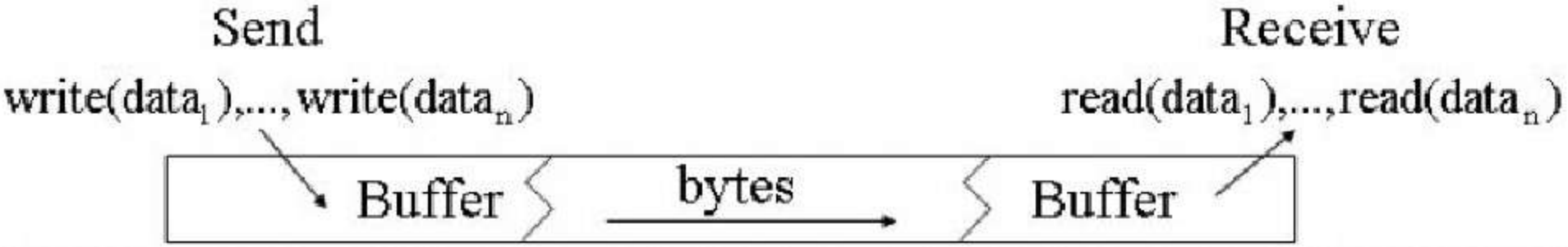
Currently do this manually using code assertions


public void write(OutputStream out) throws IOException
{ ... out.write(randomBytes); ... }
Identify: randomBytes
(in message ClientHello)
2nd parameter of Random constructor called by ClientHello.write()
2nd parameter of ClientHello constructor
public void write(OutputStream out) throws IOException
{ ... random.write(out); ... }
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

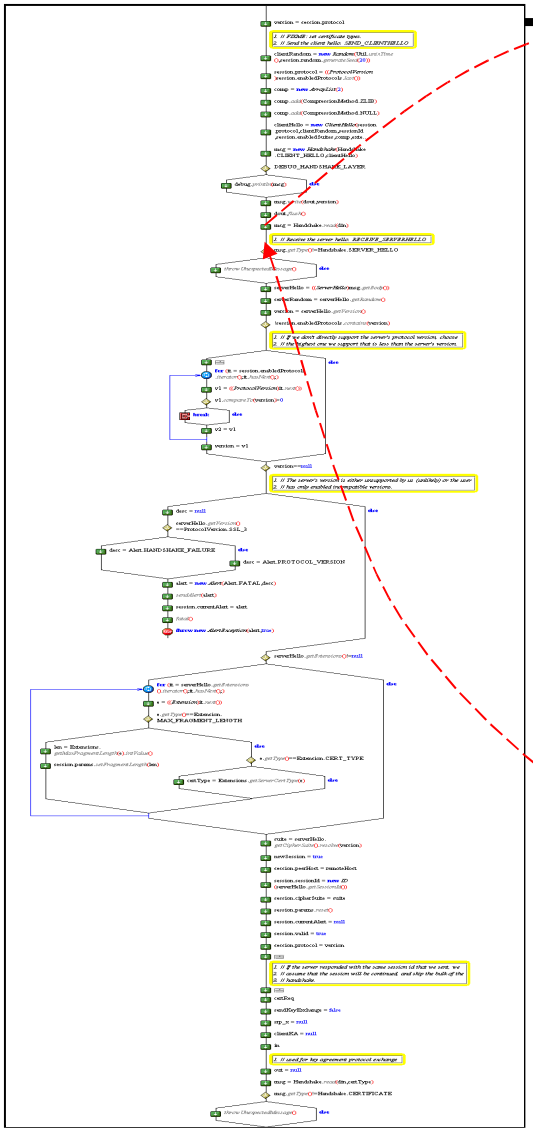
To extract input/output labels for state machine transitions, analyze input / output mechanism used in the implementation.

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).

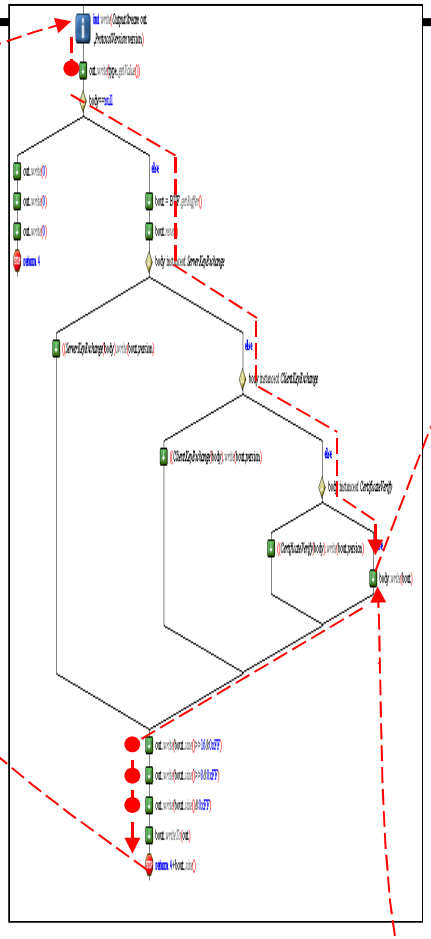


Sending Messages

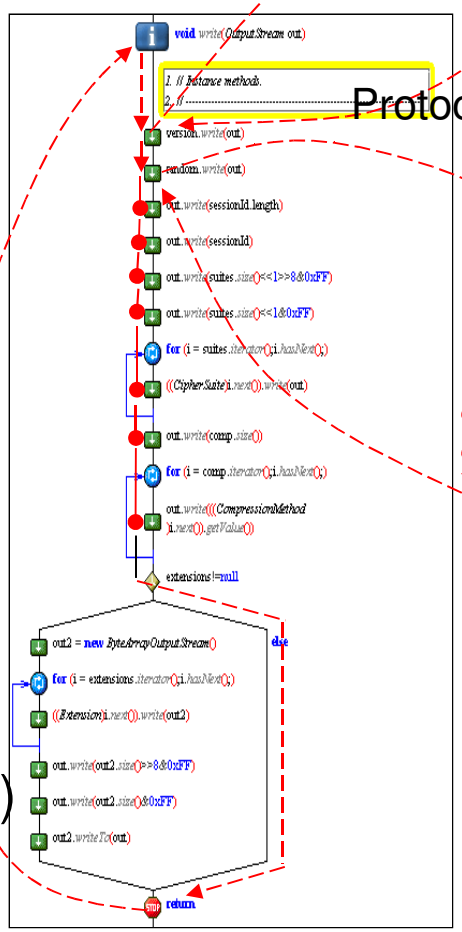
Automate this using patterns



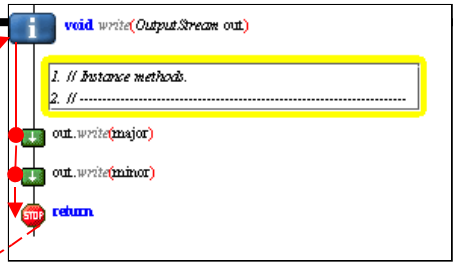
SSLSocket.doClientHandshake()



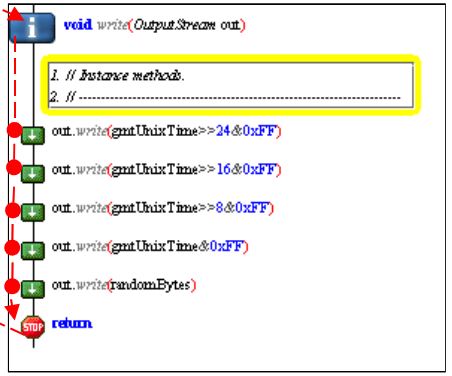
Handshake.write()



ClientHello.write()



ProtocolVersion.write()

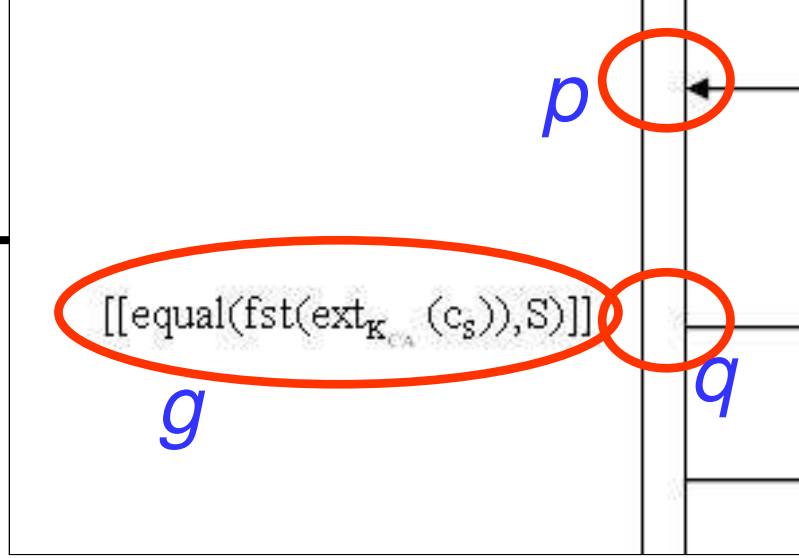


Random.write()

traverse CFG

call of OutputStream.write()

Checking Guards



Guard *g* enforced by code?

- Generate runtime check for *g* at *q* from diagram: simple + effective, but performance penalty.
- Testing against checks (symbolic crypto for inequalities). [ICFEM02]
- 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

„meaning
“

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

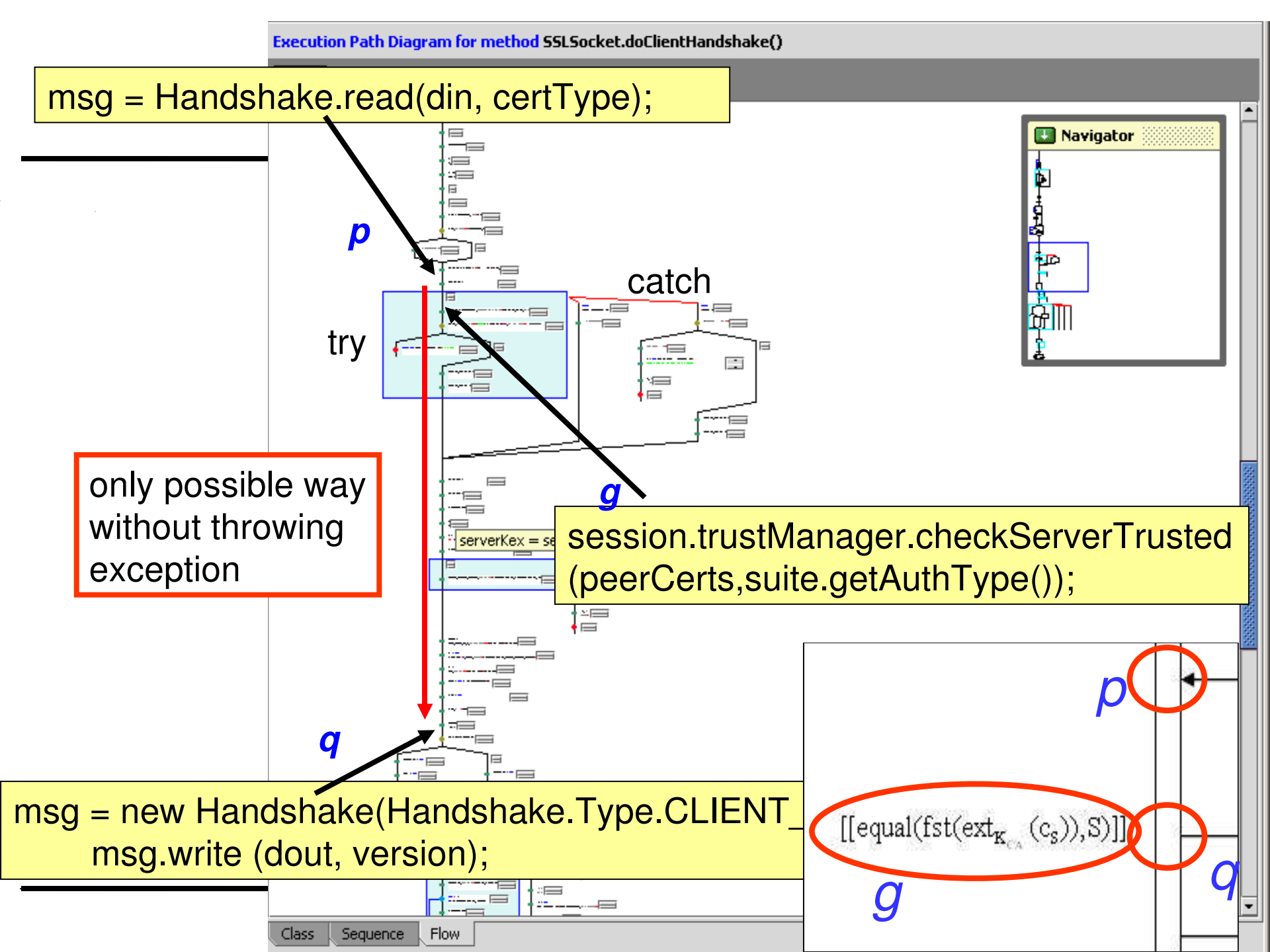
calls doVerify()

java.security.Signature

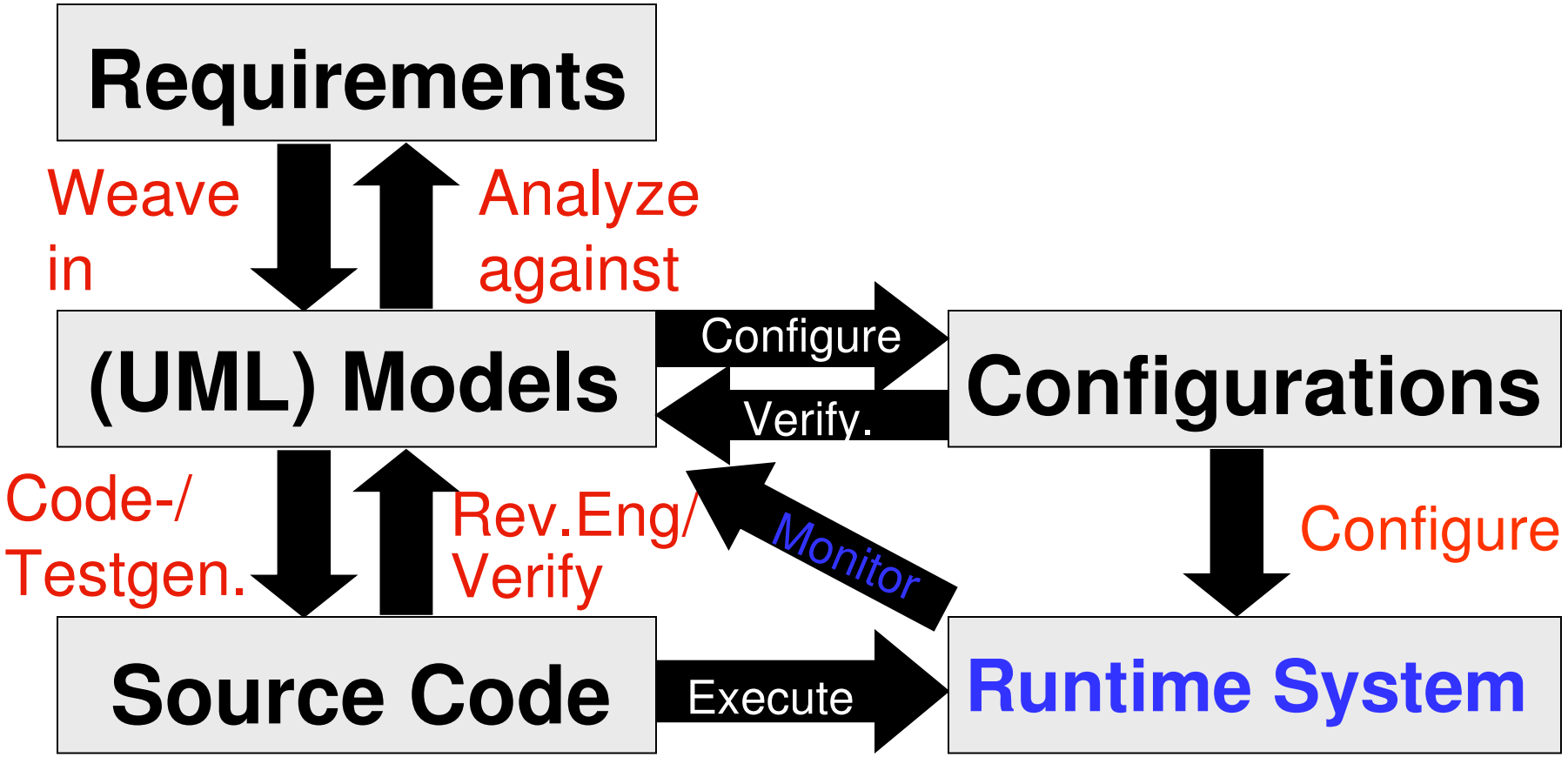
- **Initializatize**
- **Update**
- **Verify**

„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());
q
msg = new Handshake(Handshake.Type.CLIENT_
msg.write (dout, version);
[[equal(fst(ext
Class
Sequence
Flow

Roadmap


Requirements
Weave in
Analyze against
(UML) Models
Configure
Verify.
Configurations
Code-/ Testgen.
Rev.Eng/ Verify
Monitor
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 ?

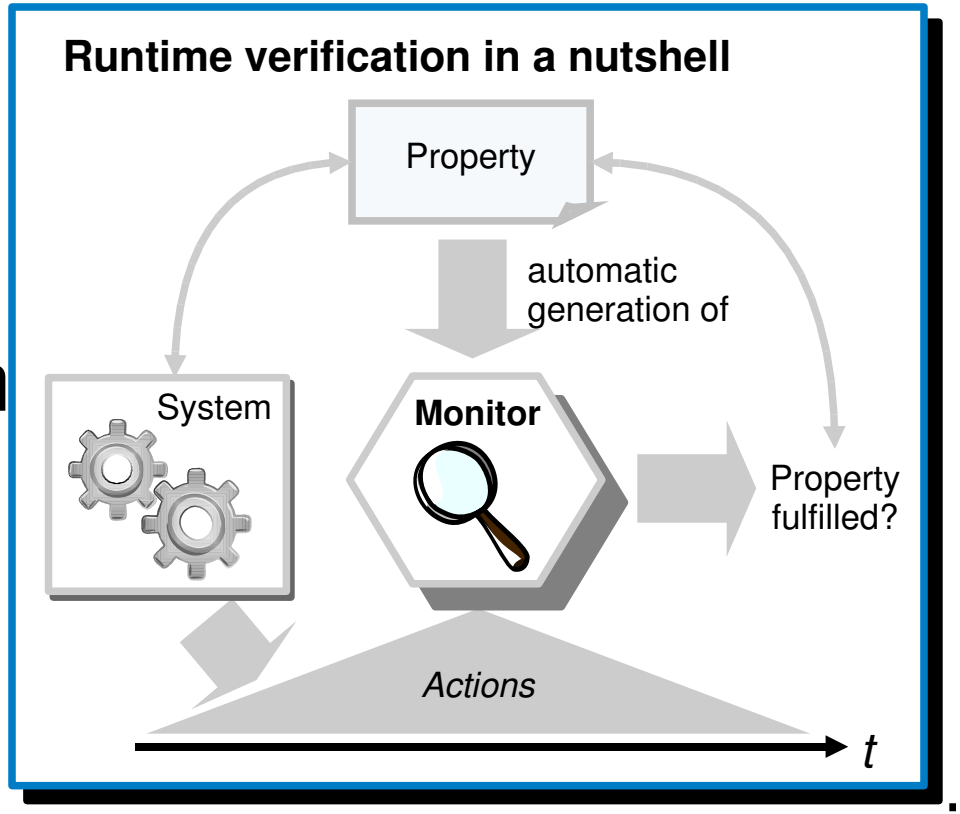
➔ Run-time verification.

Runtime Verification using Monitors

Dynamic verification technique on the actual system.

Essentially a symbiosis of model-checking and testing.

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

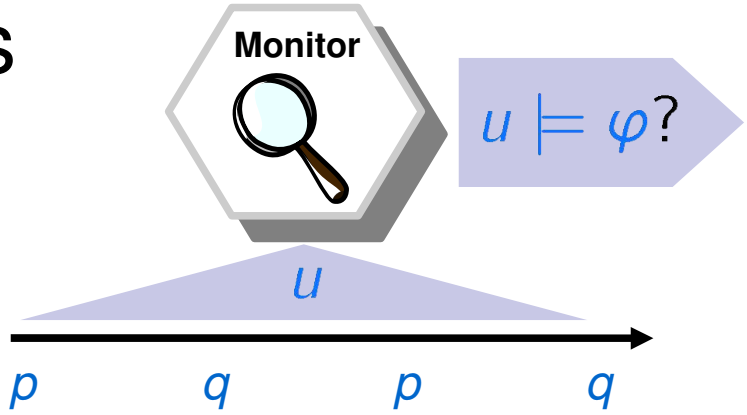


Formal underpinnings

- 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*

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

Semantics

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

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

(w word, i position)

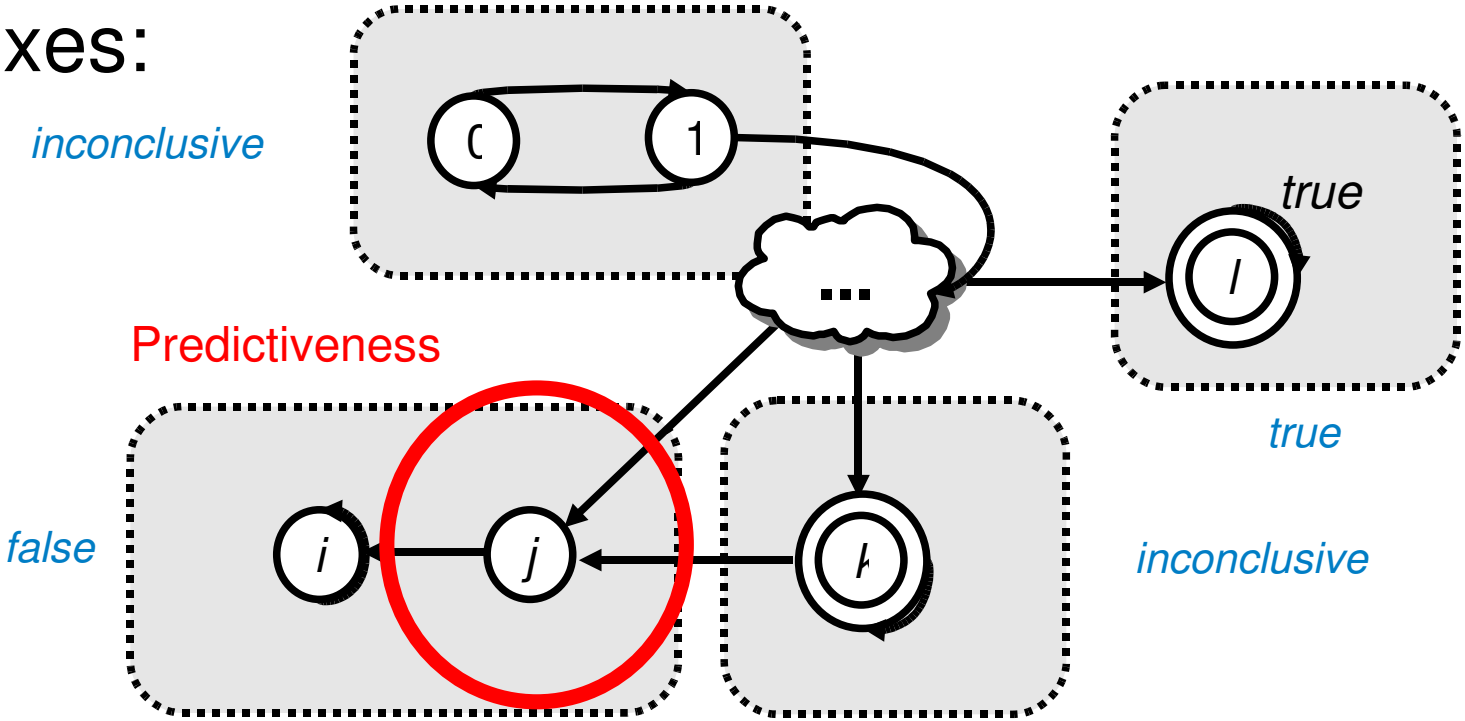
Write F phi for true U phi (“eventually phi”); G phi for not F not phi (“globally phi”); phi1 W phi2 for G phi1 or (phi1 U phi2) (weak-until)

Monitoring-friendly LTL semantics

3-valued semantics:

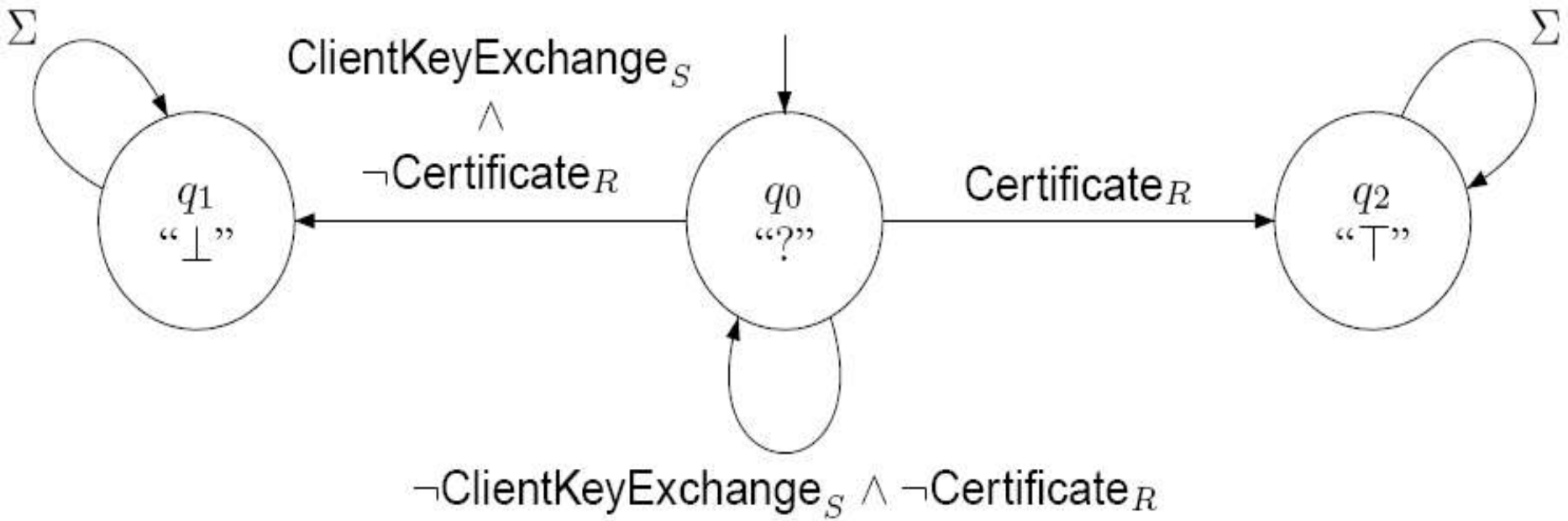
$$[u \models \varphi] = \begin{cases} \top & \text{if } \forall \sigma \in \Sigma^\omega : u\sigma \models \varphi \\ \bot & \text{if } \forall \sigma \in \Sigma^\omega : u\sigma \not\models \varphi \\ ? & \text{otherwise} \end{cases}$$

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 $\neg$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

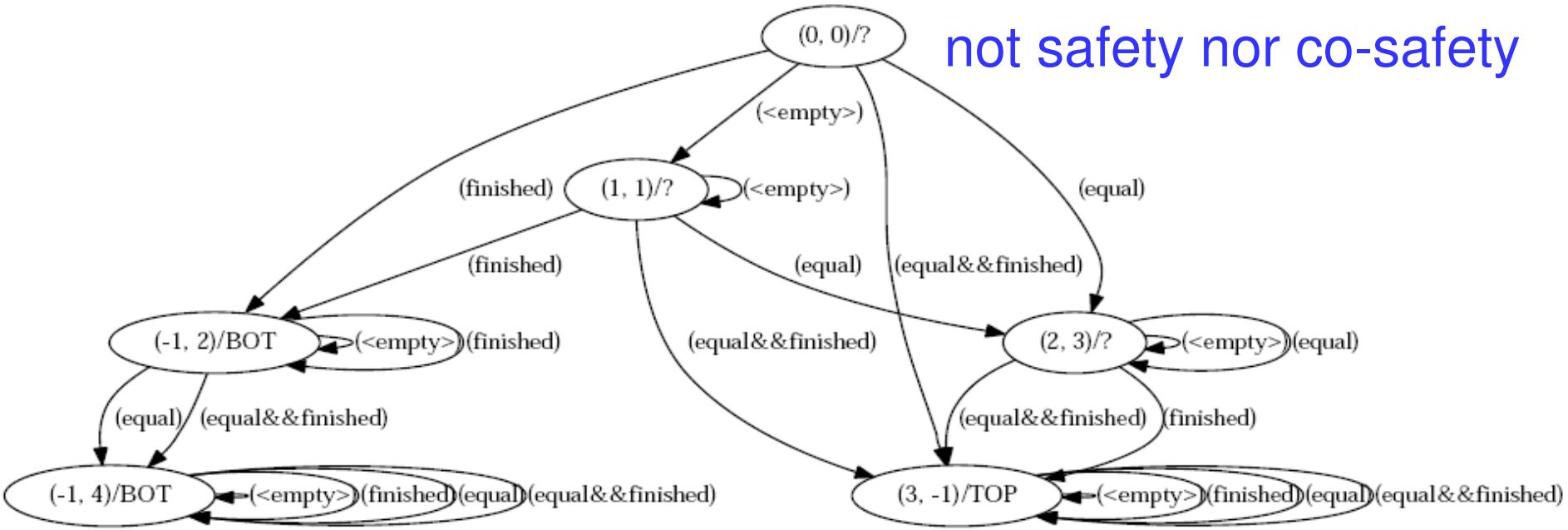
Server Finished

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



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]
- For legacy systems: need to extract security domain knowledge from the code.
  [CSMR 07, CSMR 08, IPCP 08]

Maintaining Traceability under Evolution

Code evolution can potentially multiply testing effort.

Need to re-verify code parts that changed, re-do integration testing etc.

Particular problem when testing for sophisticated properties (such as security) since requires particular effort.

➔ Want to automatically trace code evolution to model level so can automatically reuse earlier tests.

[Bauer, Jurjens, Yu 09]

Model-Code Co-Evolution

Basic observation: Most system changes can be reduced to two kinds:

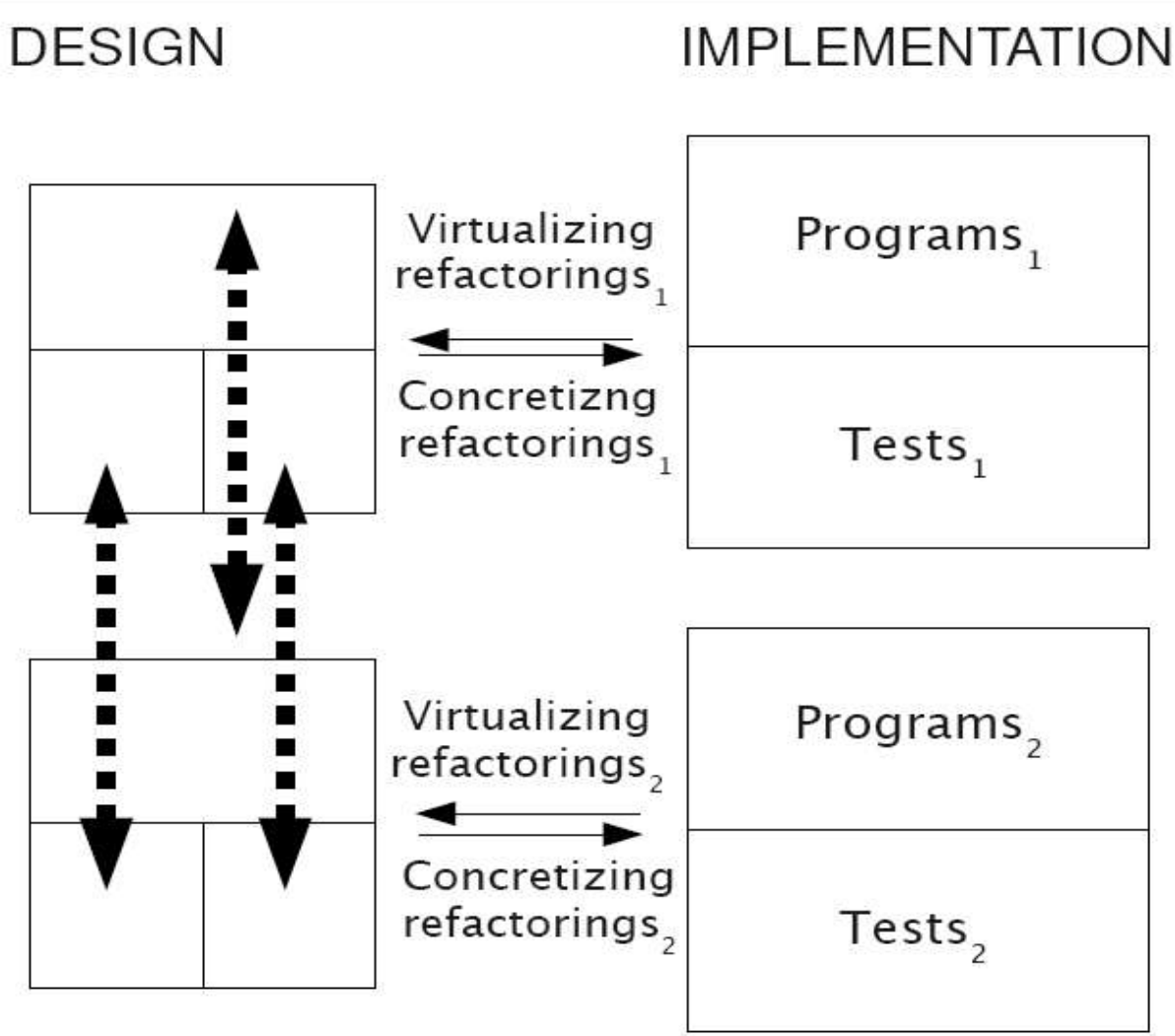
- Adding / removing parts of the system.
- Basic refactoring operations to hold system parts together despite changes.

When adding / removing code parts we need to assume that the corresponding models are also added / removed.

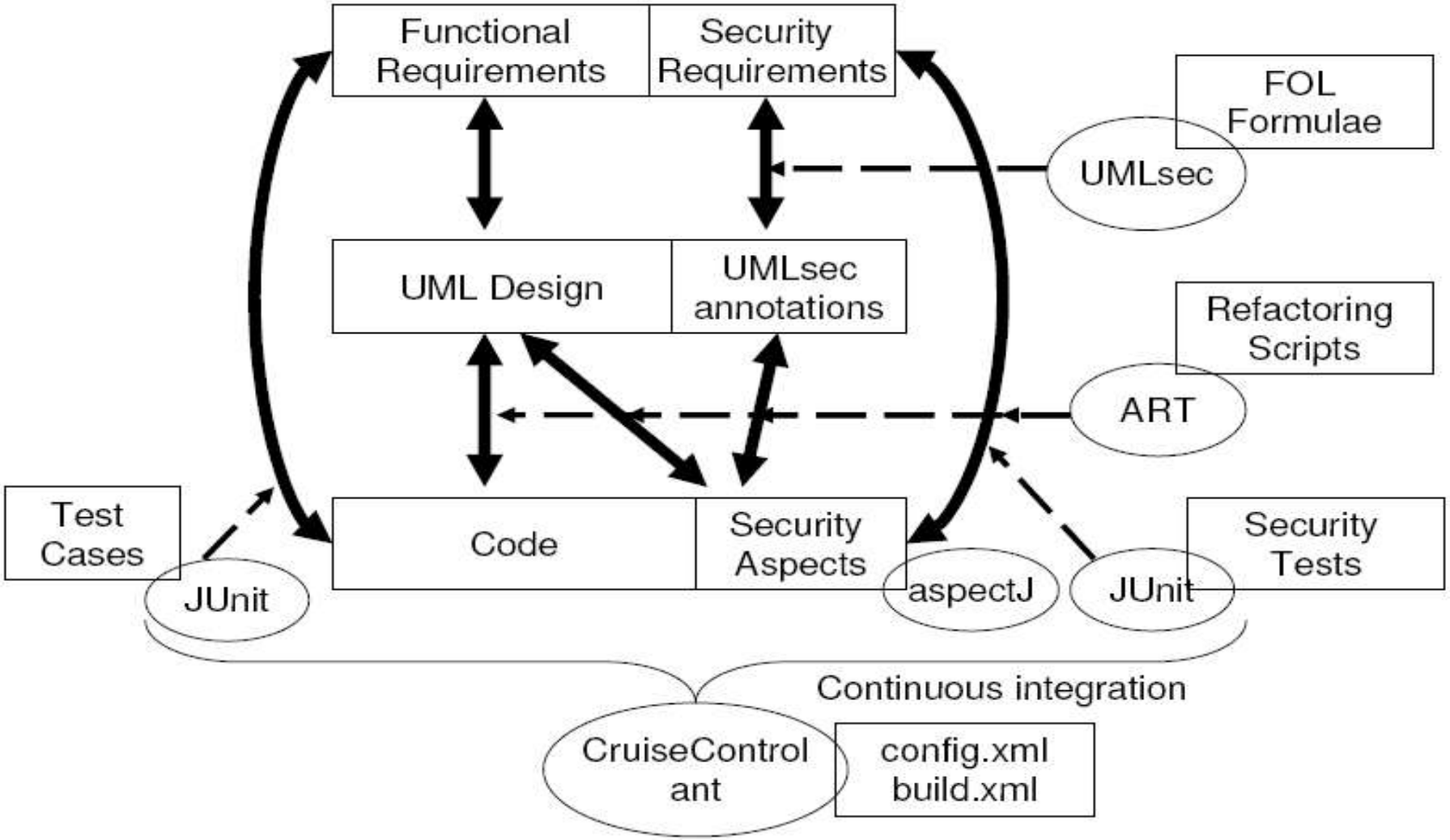
For evolution by refactoring can achieve automated model-code traceability e.g. using Eclipse Refactoring Language Toolkit (LTK) / XML based refactoring scripts.

Maintain model-code synchrony using continuous integration scripts (with CruiseControl / Apache Ant).

Evolution as Code Refactoring



Refactoring: Tool Support



Java Secure Sockets Extension

Applied our approach to a series of implementations of the Java Secure Sockets Extension library:

- Jessie 1.0.0
- Jessie 1.0.1
- JSSE 1.6

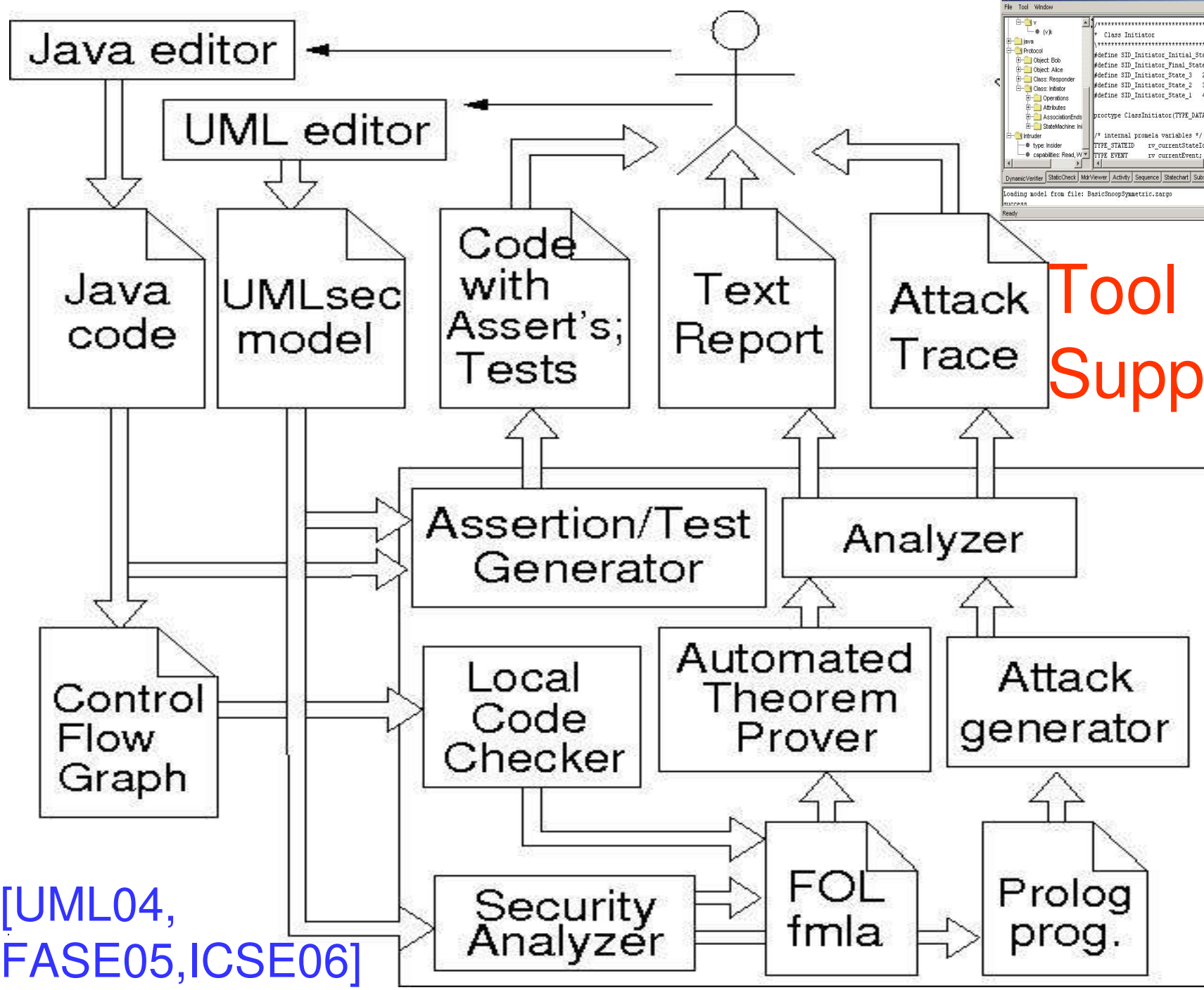
Demonstrated that our model-code co-evolution approach is robust even across major software changes.

Refactoring JSSE

Symbols	Program entities	Identif.	Refactoring op.
1. C	clientHello	C	rename.type
2. S	serverHello	S	rename.type
3. P_{ver}	session.protocol version	P_ver	extract.temp
4. R_C R_S	clientRandom serverRandom	R_C R_S	rename.local.variable rename.local.variable
5. S_{id}	sessionId sessionId	S_id S_id	rename.field rename.local.variable
6. Ciph[ ]	session.enabledSuites	Ciph	extract.temp
7. Comp[ ]	comp	Comp	extract.temp
8. Veri	Lines 1518–1557	Veri	extract.method

Refactoring: Performance

Messages in sequence	op.	diff	Time (sec)
S1: $C \rightarrow S : (P_{\text{ver}}, R_C, S_{\text{id}}, \text{Ciph}[\,], \text{Comp}[\,])$	7	31	13.891
S2. $S \rightarrow C : (P_{\text{ver}}, R_S, S_{\text{id}}, \text{Ciph}[\,], \text{Comp}[\,])$	5	20	9.437
S3. $S \rightarrow C : \text{Certificate}[\text{X509Cert}_s]$	2	2	1.474
S4. $C : \text{Veri}(\text{X509Cert}_s)$	2	2	3.854
...	...	...	...
Total of 7 messages and 3 checks	27	86	40.303


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]

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.

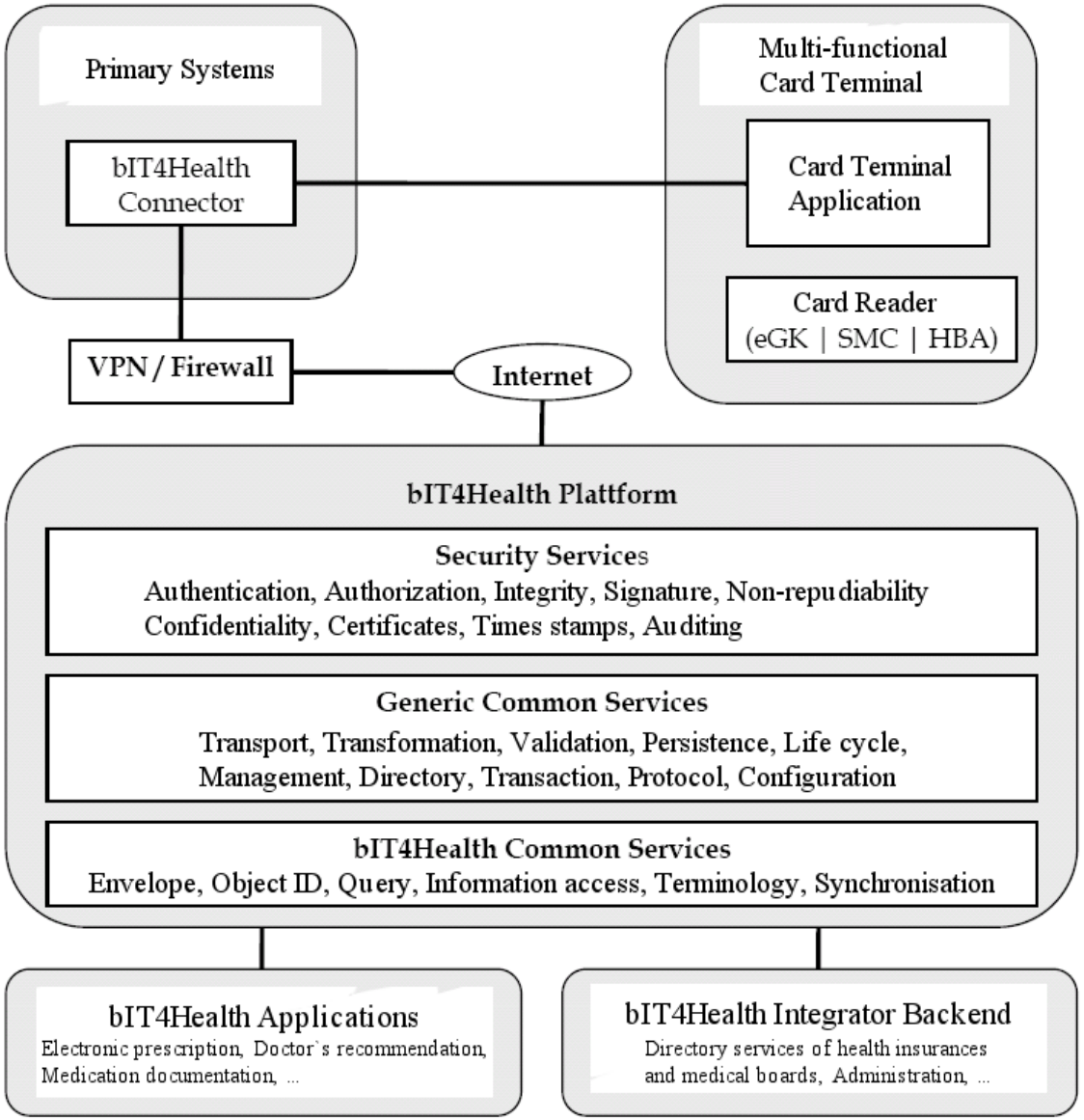
T··Systems·
Allianz
Deutsche Bank
HypoVereinsbank
CEPS™
BMW Group
msg systems
Münchener Rück
Munich Re Group

Bundesministerium für Bildung und Forschung
Bundesministerium der Verteidigung
O2
infineon
Bundesministerium für Wirtschaft und Technologie

German Health Card Architecture

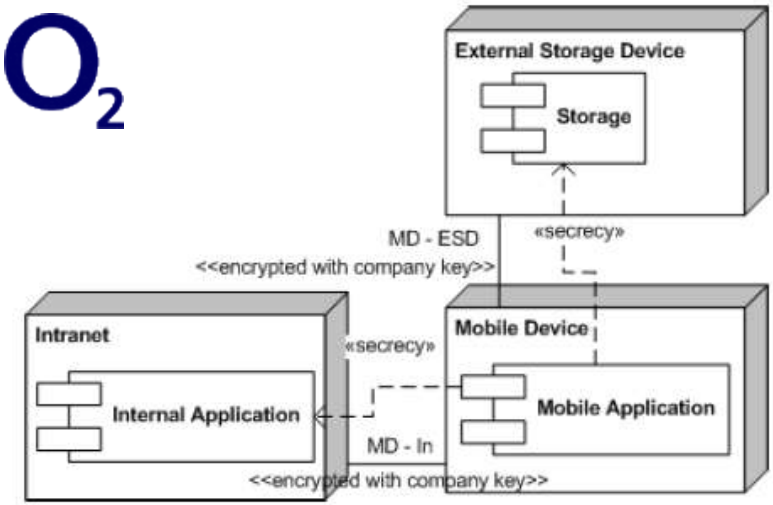
- Analyzed architecture against security requirements using UMLsec
- Detected several security weaknesses in the architecture

[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.



Intranet Information System

[ICSE 07]

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.

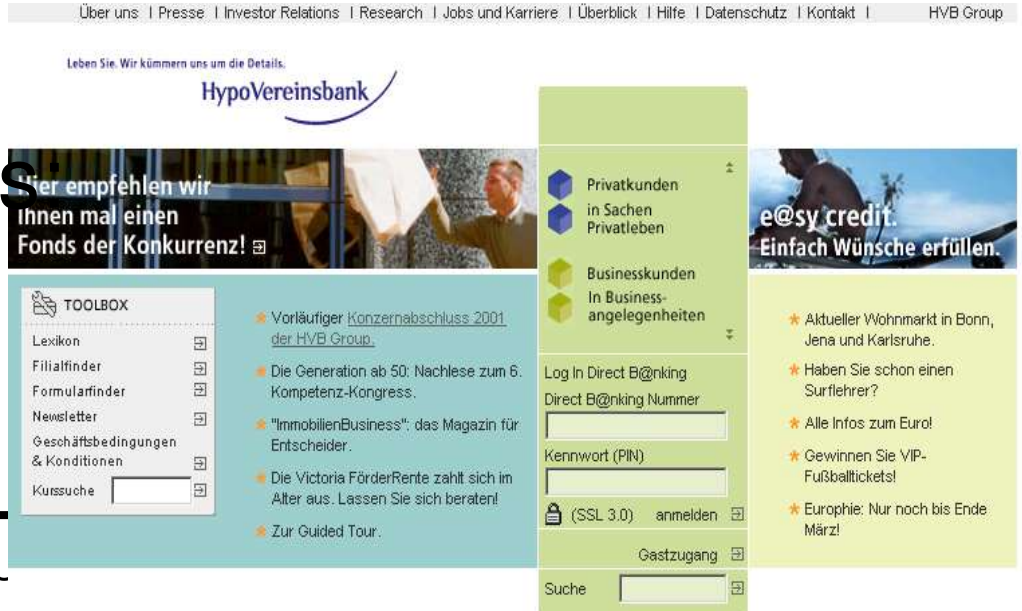
Bank Application

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

Conclusions

Model-based Security Engineering using UMLsec:

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