Security Certification in the Presence of Evolution: Models vs. Code

Jan Jürjens

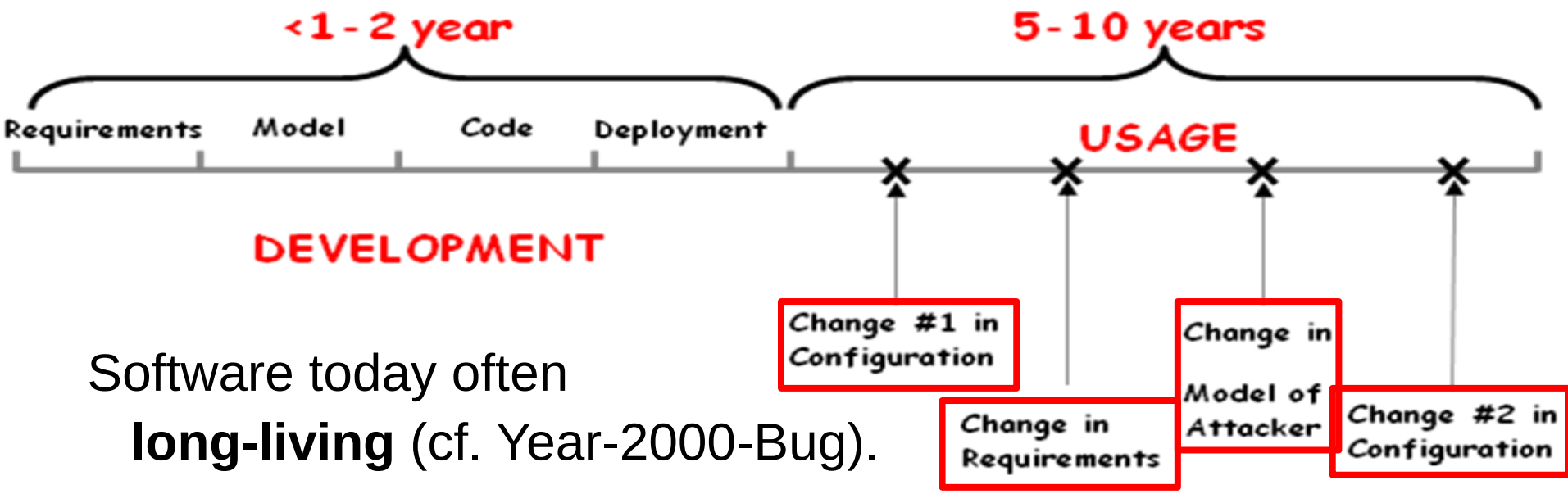
TU Dortmund & Fraunhofer ISST

http://jan.jurjens.de

technische universität dortmund

Fraunhofer ISST

What is Software Evolution ?



Software today often
long-living (cf. Year-2000-Bug).

Continual change during lifetime.

- Changing **requirements**, changing **environment**
- **Bug** fixing

➔ **Software evolution !**

What is the Challenge about Software Evolution ?

After each change: redo software tests (**regression test**).

High costs:

- E.g. **Y2k-problem**:
  Ca. 600 Bill. US$ world-wide.

➔ Often **insufficient regression tests**:

- Explosion of **Ariane 5** (1996):
  Software from Ariane 4 reused in Ariane 5 without **sufficient regression** tests.
  ➔ 370 Mio US$ damage.

How to solve this Problem?

Goal: Integrate **quality assurance** with **evolution**:

- **Reuse** QA results as far as possible to reduce costs.

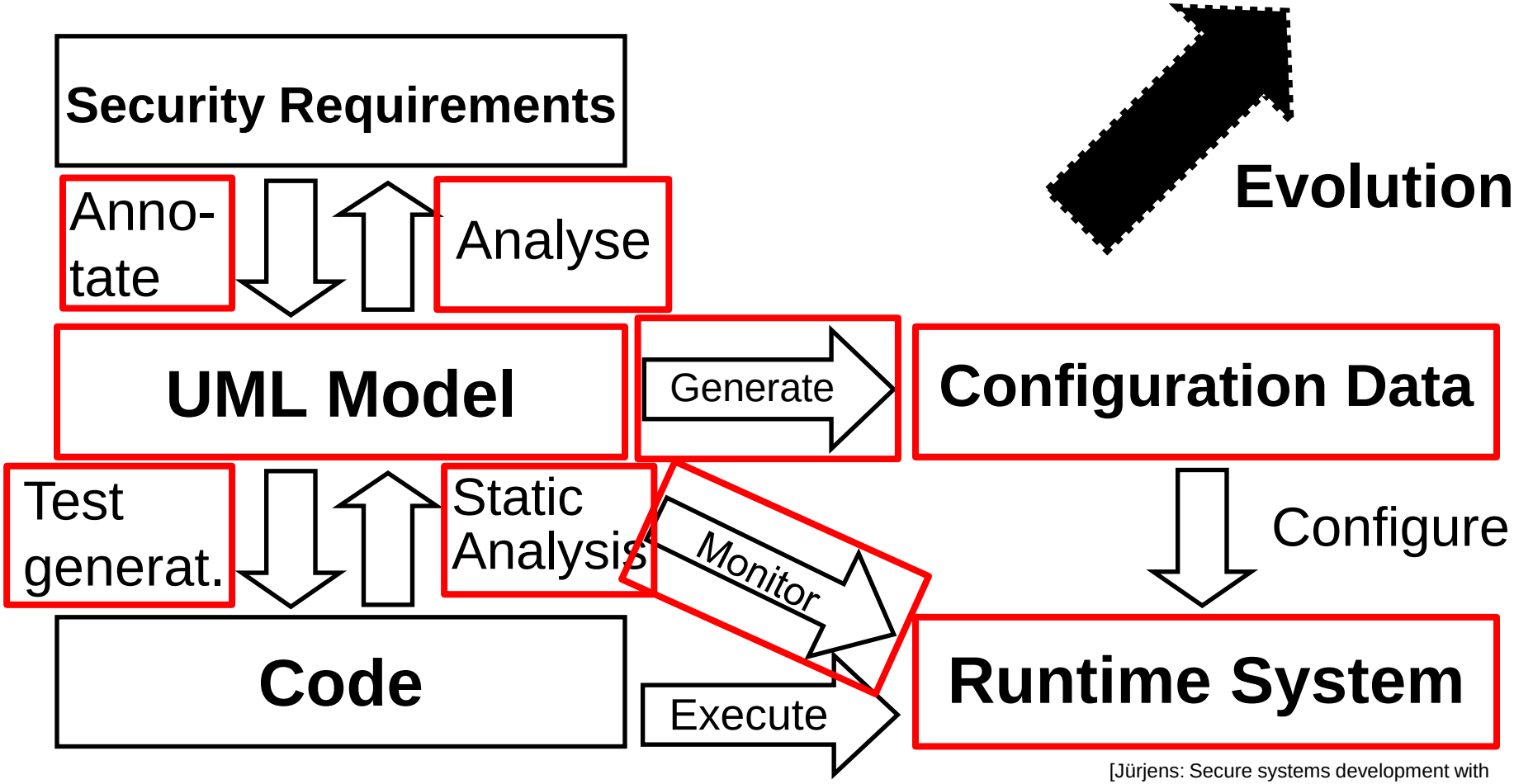
➔ Under which **conditions requirements preserved** ?

- Only re-verify when conditions **not** fulfilled.
  - And only systems parts where **necessary**.
- Check at model level **before** evolution **carried out**.

Secure Software Evolution

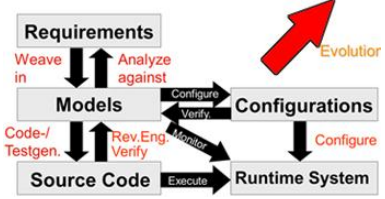
- Challenge and Scientific Basis
- Secure Software Evolution
  - General Approach
  - Preservation Results
- Validation

Model-based Security Assurance



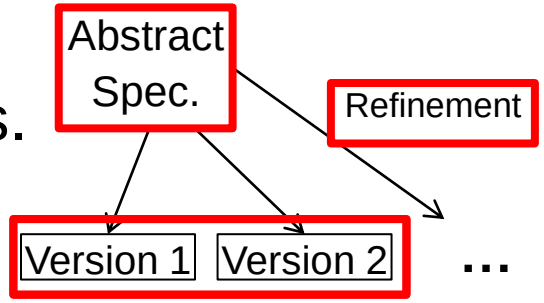
[Jürjens: Secure systems development with UML. Springer 2005. Chines. Übers. 2009]

Design Techniques / Architecture Principles for Management of Evolution



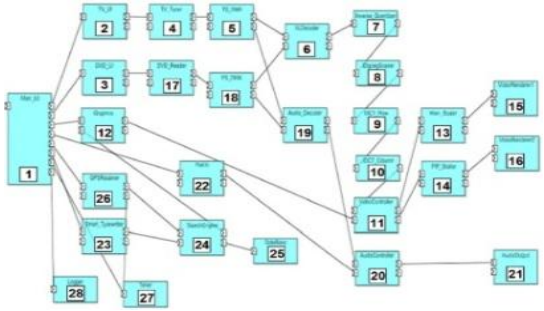
Refinement of specifications

➔ Evolution between different refinements.



Applying refactorings

➔ Carry out evolution in controllable way.



Applying design patterns

➔ Reduce complexity of evolution.

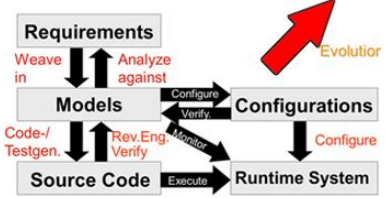
Architectural principle modularization

➔ Limit impact of change to system parts.

Question: when do these preserve security properties ?

[Taubenberger, Jürjens, Yu, Nuseibeh. Resolving Vulnerability Identification Errors using Security Requirements on Business Process Models. Journal on Information Management and Computer Security, 2013.]

Evolution vs. Design / Architecture When Properties Preserved ?



Refinement: Developed refinement which **preserves** security.

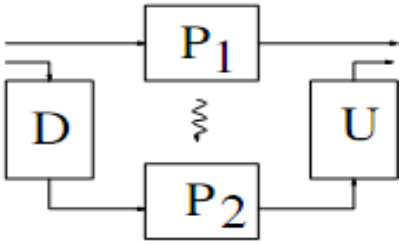
Conditions for preservation of security through…

… Refactoring: using **Eclipse refactoring scripts**

… Applying design patterns: „Gang of Four“ patterns

… Modularization:

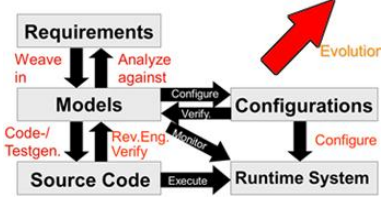
- Layering of **architectural levels**
- **Component**-oriented architectures
- **Service**-oriented architectures
- **Aspect**-oriented development



Experiments: Relevant in 57% of code base.

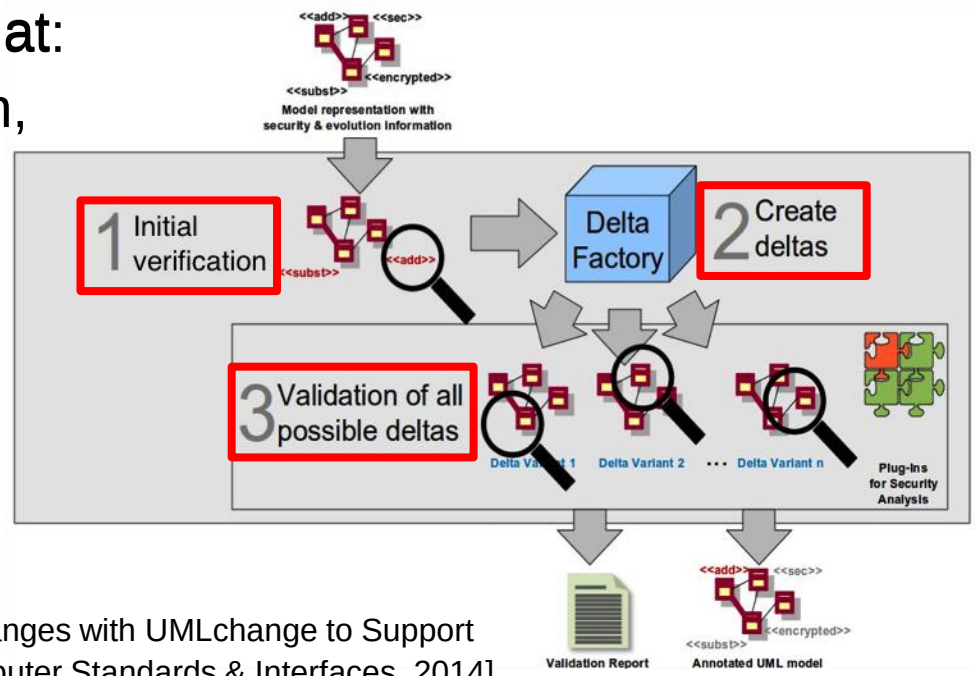
[Felderer, Katt, Kalb, Jürjens et al.: Evolution of Security Engineering Artifacts. Int. Journal of Secure Software Engin. (IJSSE), 2014

Evolution-based Verification



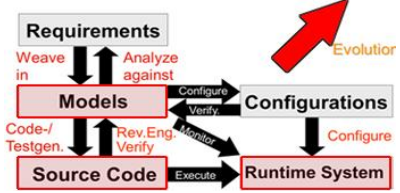
- Initial verification: Register which **system parts** relevant.
- Store partial results in model („**proof-carrying models**“).
- **Compute difference:** old vs. new model (e.g. with SiDiff [Kelter]).
- Only re-verify **system parts** that:
  1) **relevant** in initial verification,
  2) **changed,** such that
  3) conditions on **preservation** of security **not fulfilled**.

Significant speed-up vs. **complete** re-verification.



[Wenzel, Warzecha, Jürjens, Ochoa. Specifying Model Changes with UMLchange to Support Security Verification of Potential Evolution. Journal of Computer Standards & Interfaces, 2014]

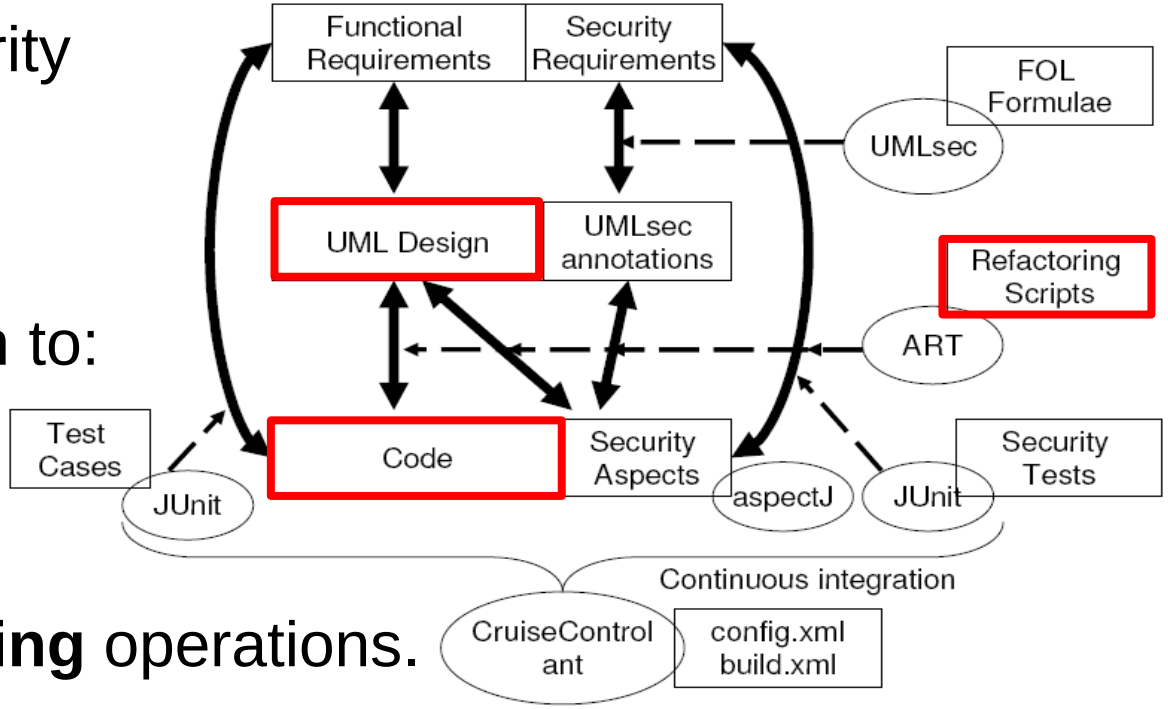
Model-Code Traceability at Evolution



Goal: Preserve model-code **traceability** of security properties during **evolution**.

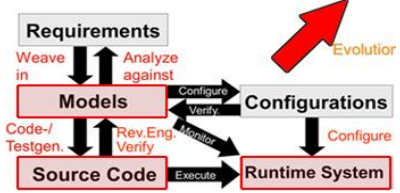
Idea: Reduce evolution to:

- **Adding / deleting** system elements.
- Supporting **refactoring** operations.



➔ Approach for automated model-code traceability based on **refactoring scripts** in Eclipse.

Security Analysis at Implementation Level



Vulnerability in **OpenSSL:**

(CVE-2008-5077, 7.1.2009, http://www.openssl.org/news/vulnerabilities.html)

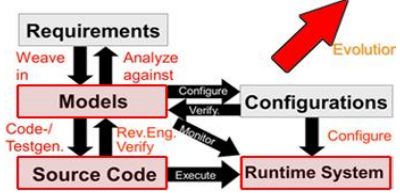
"Several functions inside OpenSSL **incorrectly checked** the result after calling the EVP_VerifyFinal function, allowing a **malformed** signature to be treated as a **good signature** rather than as an error."

```
int check_certificate(){
  ...
  if(certificate_malformed)
    return -1;
  else if(!certificate_check_ok)
    return 0;
  else return 1;}
```

```
// later in code:
...
if(check_certificate()) // oops!
{
  trust_certificate();
}
```

Feb/Mar 2014: "**goto fail**"-vulnerabilities in SSL @ iPhone, GnuTLS.

Security Analysis by Symbolic Execution



Compile protocol implementation to **symbolic model** for security analysis.

Example message: $A \xrightarrow{m, hash(m, k_{shared})} B$

C code:

```
client(char * payload, int payload_len){
  int msg_len = 5 + len + SHA1_LEN;
  char * msg = malloc(msg_len);
  char * p = msg;
  *p = len; p += 4;            // add length
  *(p++) = 1;                  // add the tag
  memcpy(payload, p, len);     // add the payload
  sha1(msg, 5 + len, p);       // add the hash
  send(msg, msg_len);          // send
}
```

Model from **symbolic execution**:

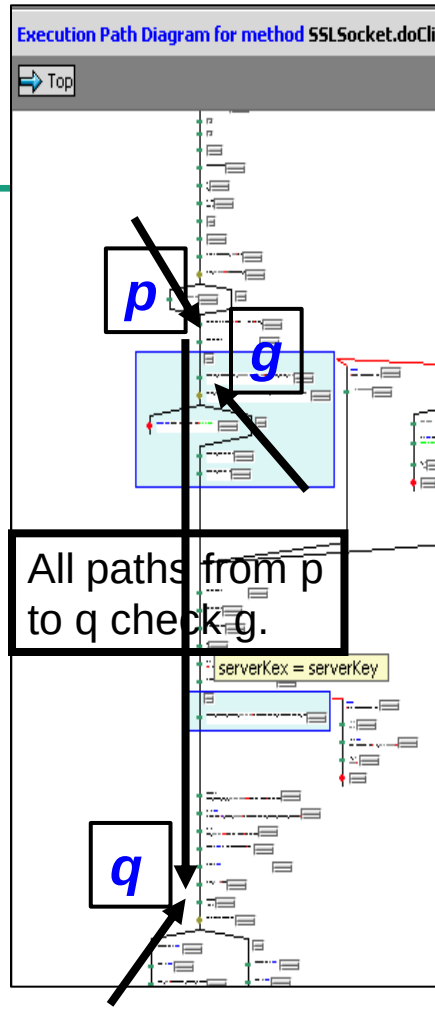
$$\mathbf{out}(\mathbf{len}(\text{payload}) \mid 01 \mid \text{payload} \mid \text{sha1}(\mathbf{len}(\text{payload}) \mid 01 \mid \text{payload}, k_{shared}))$$

Code-level Security Verification Subject to Evolution

Project Csec (with Microsoft Research Cambridge):

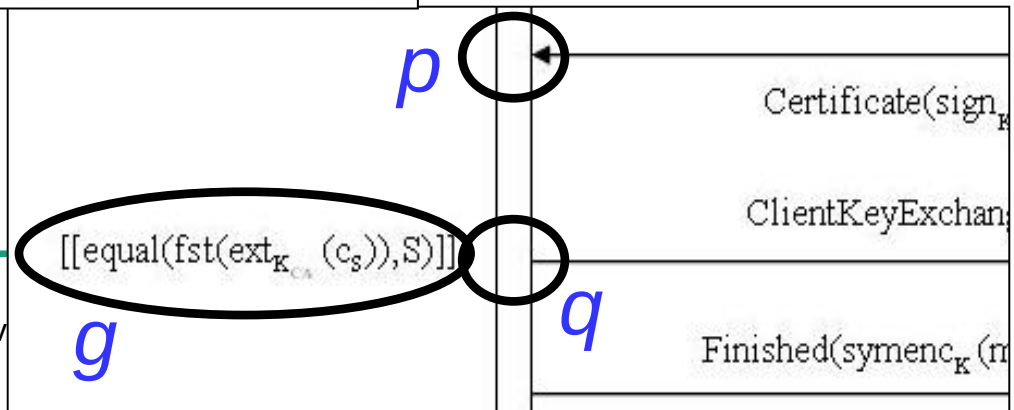
Static code analysis against **security** properties.

Evolution-based model analysis + model-code traceability ➔ evolution-sensitive **static code security analysis**.

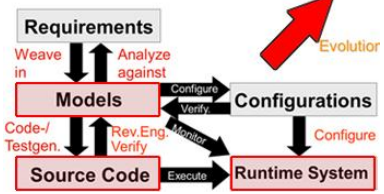


	C LOC	IML LOC	outcome	result type	time
simple mac	~ 250	12	verified	symbolic	4s
RPC	~ 600	35	verified	symbolic	5s
NSL	~ 450	40	verified	computat.	5s
CSur	~ 600	20	flaws found	—	5s
Metering	~ 1000	51	flaws found	—	15s

[Dupressoir, Gordon, Jürjens, Naumann: Guiding a General-Purpose C Verifier to Prove Cryptographic Protocols. Journal of Computer Security 2014]



Run-time Verification subject to Evolution



Source code not available ➔ run-time monitoring.

Relevant approach: Security Automata [F.B. Schneider 2000].

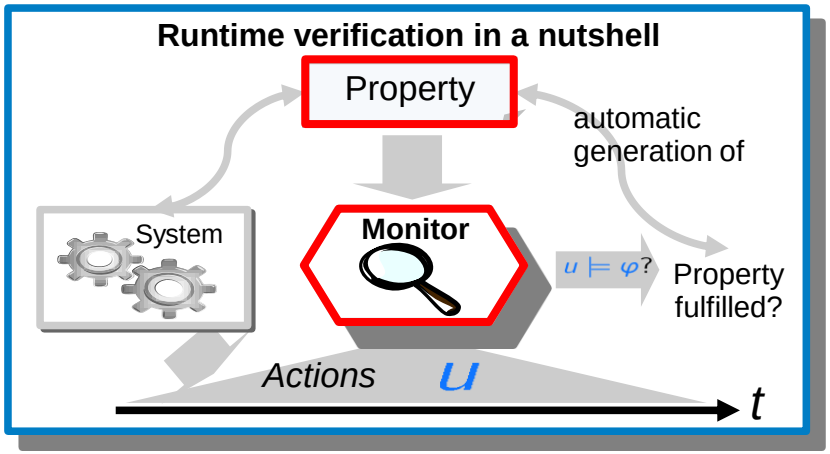
Problem: no evolution, only **„safety"-properties**.

➔ New approach, based on runtime verification[1]:

[1] Havelund, Grosu 2002

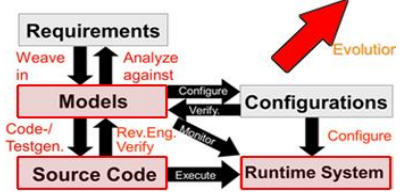
- Security property in LTL.
- Generate monitors with **evolution-based traceability**.

Now also **non-safety-properties** (using 3-valued LTL-semantics).



Client	Server	No Monitor [s]	Monitor [s]	Overhead [s]	Overhead [%]
GnuTLS	GnuTLS	0.109	0.120	0.011	10.313
OpenSSL	JESSIE	0.158	0.172	0.014	8.986
GnuTLS	JESSIE	0.144	0.148	0.004	2.788

For Monitoring: 3-valued LTL-Semantics



[Bauer, Jürjens, Yu: Runtime Security Traceability for Evolving Systems. Computer Journal, 2011]

Goal for monitoring:

Detect as **early** as possible that property **will** be violated.

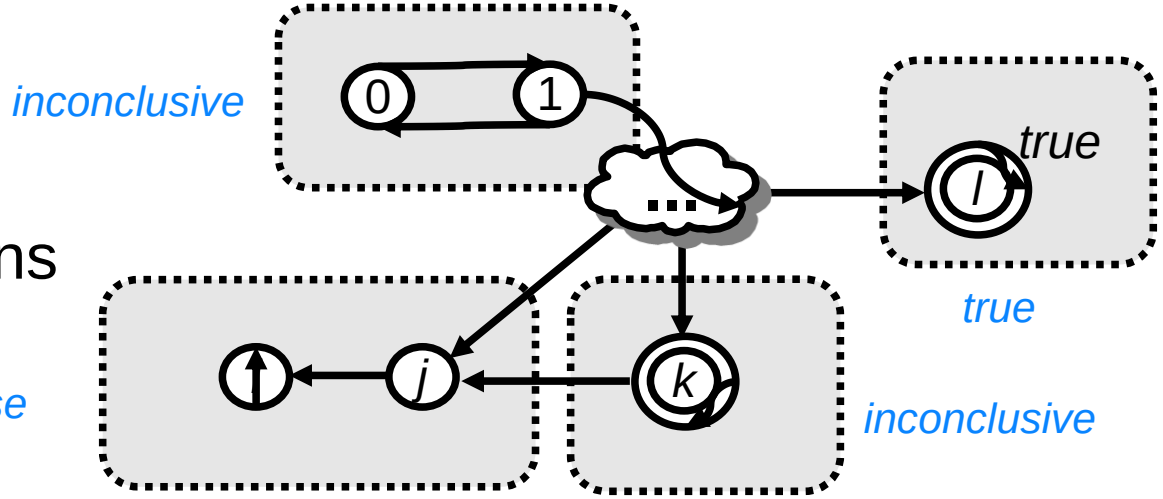
Use **3-valued semantics** for this.

➔ Finite automata for **minimal prefix** of violating state.

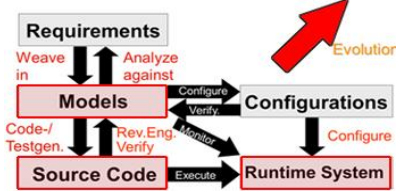
$$[u \models \varphi] = \begin{cases} \top & \text{if } \forall \sigma \in \Sigma^\omega : u\sigma \models \varphi \\ \bot & \text{if } \forall \sigma \in \Sigma^\omega : u\sigma \not\models \varphi \\ ? & \text{otherwise} \end{cases}$$

Also **non-safety-properties**:

Boolean combinations of safety and co-safety



Example Property: Server Finished



Server **sends** message Finished (event "finished") **only after** message Finished **received** from Client and MD5-hash **equals** MD5 at Server side (event "equal"). **Then** sends it out.

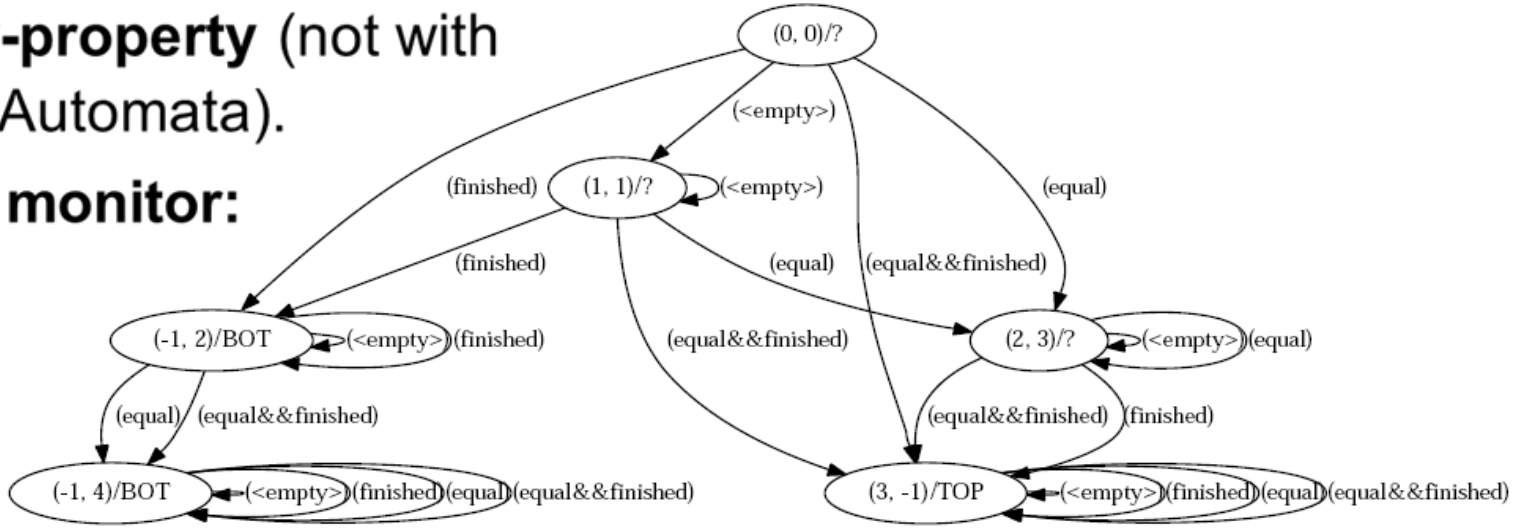
LTL formalization:

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

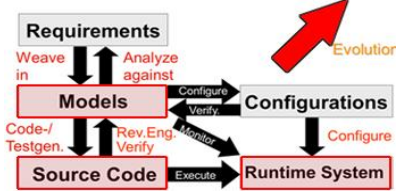
(F phi: "eventually phi"; phi1 W phi2: "phi1 weak-until phi2")

Non-safety-property (not with Security Automata).

Generated monitor:



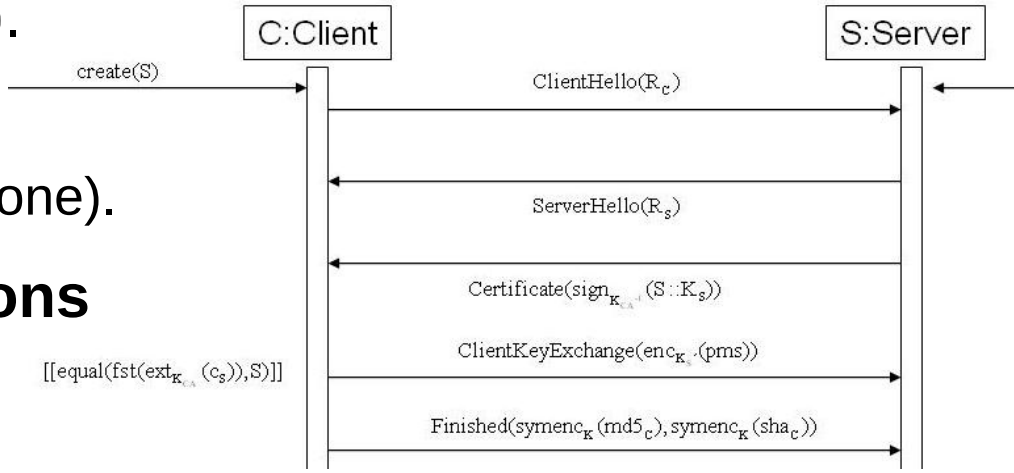
Application: Java Secure Sockets Extension



Several versions of **Java security library** "Java Secure Sockets Extension (JSSE)" and open-source re-implementation (Jessie).

Found several **weaknesses** (similar "goto fail" in SSL@iPhone).

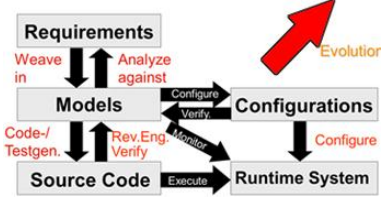
Works also for **large evolutions** (re-implementation).



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	clientRandom	R_C	rename.local.variable
R_S	serverRandom	R_S	rename.local.variable
5. S_{id}	sessionId	S_id	rename.field

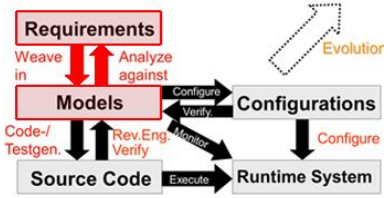
Messages in sequence	op.	diff	Time (sec)
S1: $C \rightarrow S : (P_{\text{ver}}, R_C, S_{\text{id}}, \text{Ciph}[\,], \text{Comp}[\,])$	7	31	13.891
S2. $S \rightarrow C : (P_{\text{ver}}, R_S, S_{\text{id}}, \text{Ciph}[\,], \text{Comp}[\,])$	5	20	9.437
S3. $S \rightarrow C : \text{Certificate}[\text{X509Cert}_s]$	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

Secure Software Evolution



- Challenge and Scientific Basis ✓
- Secure Software Evolution
  - General Approach ✓
  - Preservation Results
- Validation

Security Analysis: Formalization of Model



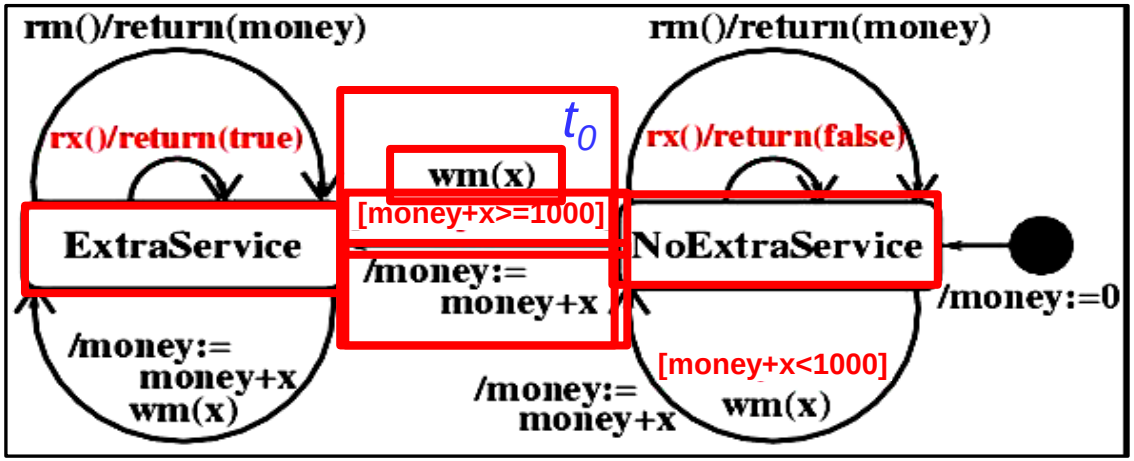
Formalize model execution. For transition $t=(source,msg,cond[msg],action[msg],target)$ and message m, execution formalized as:

$$Exec(t,m) = [\, state_{current}=source \wedge m=msg \wedge cond[m]=true \Rightarrow action[m] \wedge state_{current.t(m)}=target\,].$$

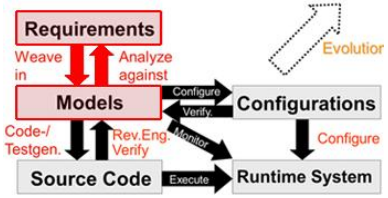
(where $state_{current}$ current state;
$state_{current.t(m)}$ state after executing t on message m).

Example: Transition t_0:

$$Exec(t_0,m)= [\, state_{current}=NoExtraService \wedge m=wm(x) \wedge money_{current}+x>=1000 \Rightarrow money_{current.t_0(m)}=money_{current}+x \wedge state_{current.t_0(m)}=ExtraService\,].$$



Security Analysis: Formalization of Property



Example **„secure information flow“:** No information flow from **confidential** to **non-confidential** data (not even partial / indirect).

Analysis: States differ only in **confidential** variables

➔ same impact on **non-confidential** variables:

$$\boxed{state_{current} \approx_{pub} state'_{current}} \Rightarrow \boxed{state_{current.t(m)} \approx_{pub} state'_{current.t(m)}}$$

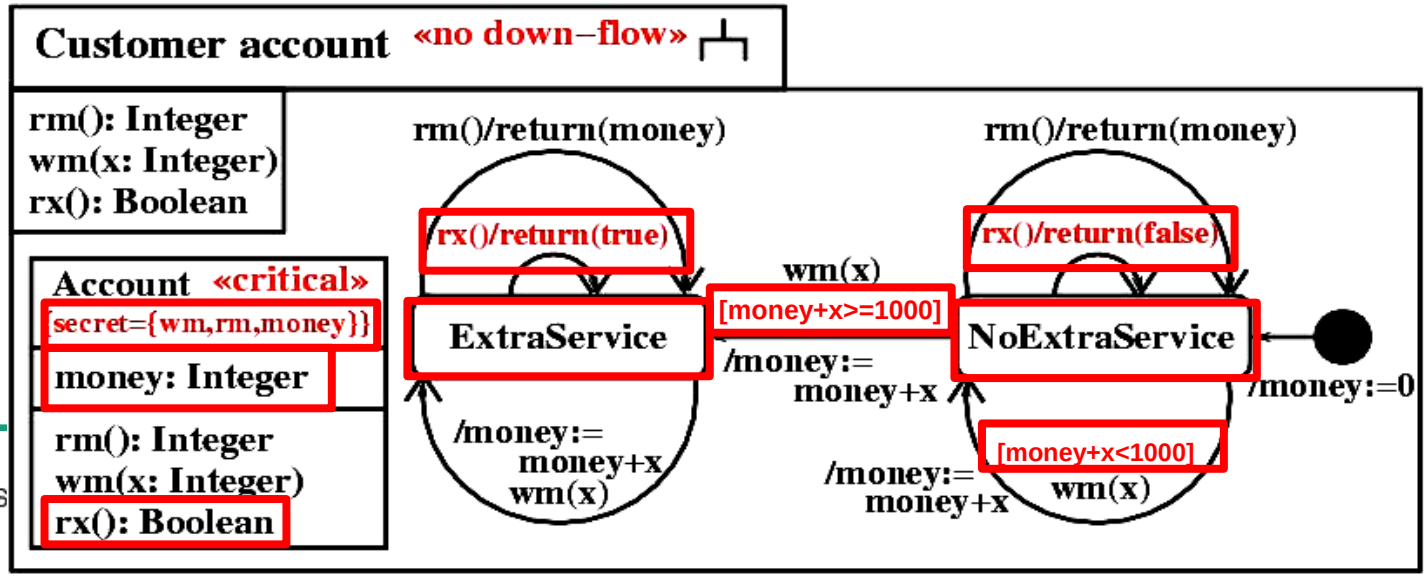
($state_{current} \approx_{pub} state'_{current}$: differ only in **confidential** variables).

Example **insecure**: Partial information flow from **confidential** attribute *money* to return value of **non-confidential** method *rx()*.

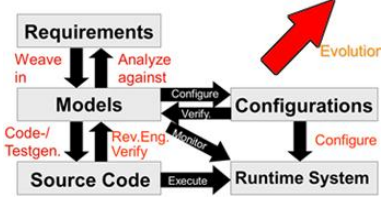
ExtraService $\approx_{pub}$ *NoExtraService*

aber nicht:

ExtraService.rx() $\approx_{pub}$ *NoExtraService.rx()*



Security vs. Refinement: Problem



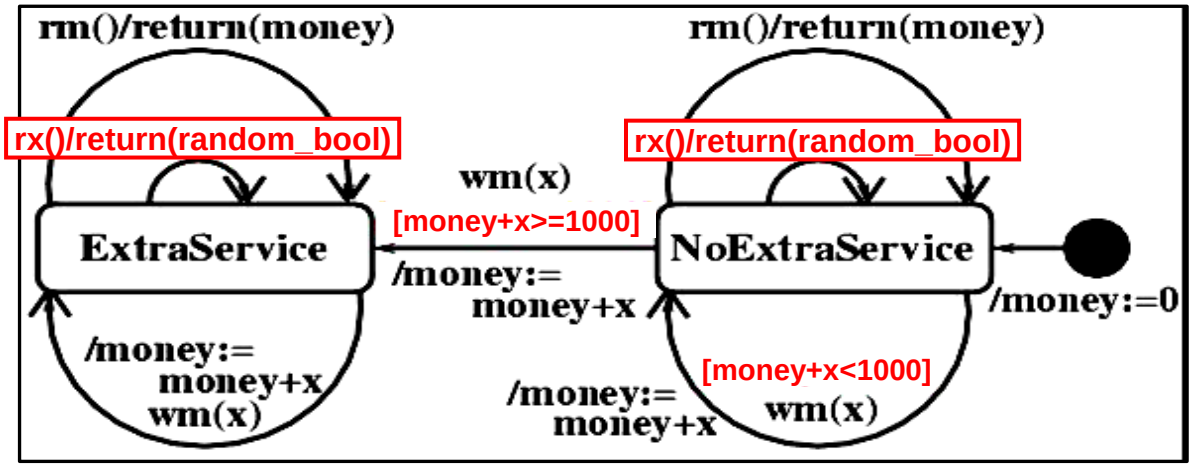
For behaviour preserving refinement:

Would expect preservation of security.

„**Refinement Paradox**“: In general **not** true !

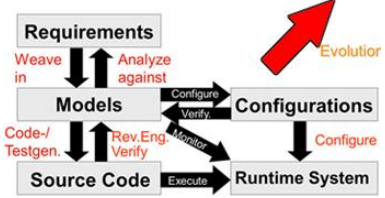
Example:

Transition *rx()/return(true)* (resp. *false*) is refinement of „secure“ transition *rx()/return(random_bool)*.



Problem: Mixing **non-determinism** as under-specification resp. as security mechanism.

Security vs. Refinement: Solution



Our specification approach **distinguishes** the two kinds of non-determinism.

Result: **Refinement** now **preserves** security requirements.

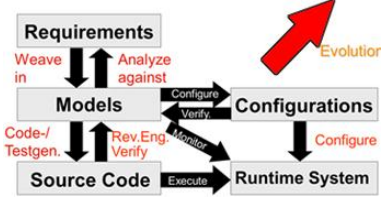
Definition Q refines P ($P \rightsquigarrow Q$) if for each $\vec{s} \in \mathbf{Stream}_{I_F}$ have $[\![P]\!](\vec{s}) \supseteq [\![Q]\!](\vec{s})$.

Theorem If P preserves secrecy of m and $P \rightsquigarrow Q$ then Q preserves secrecy of m.

Proof: using formal semantics.

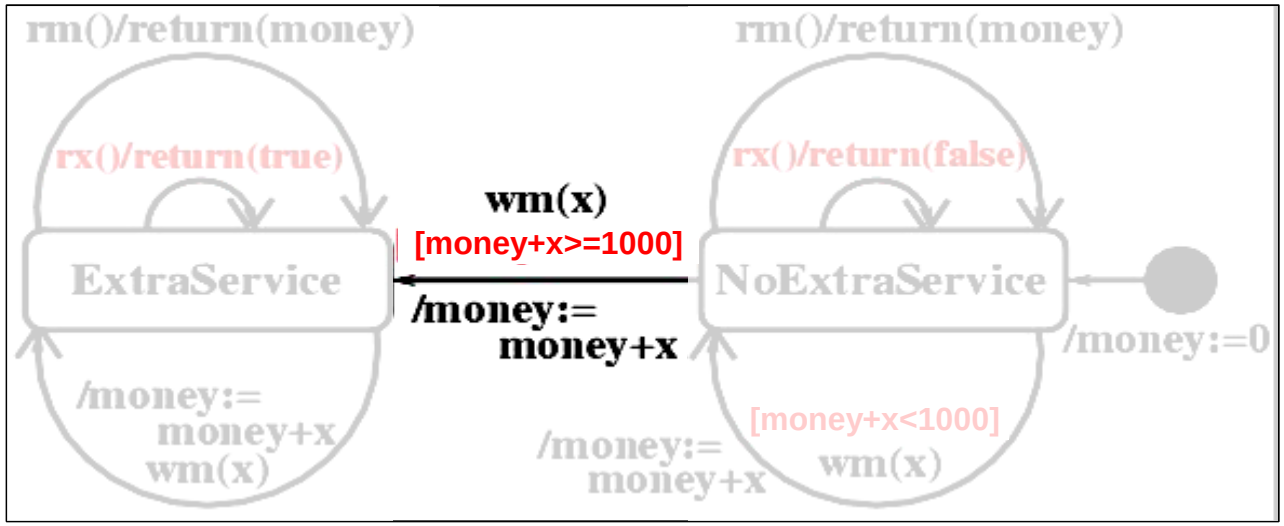
Above example: with our approach: **not** a refinement.

Security vs. Modularization: Problem



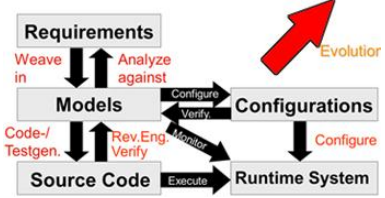
Problem: Security in general **not compositional**.

Above example: States *ExtraService* and *NoExtraService* each „secure “ (only one return value for *rx*), but **composition** in statechart **not**.



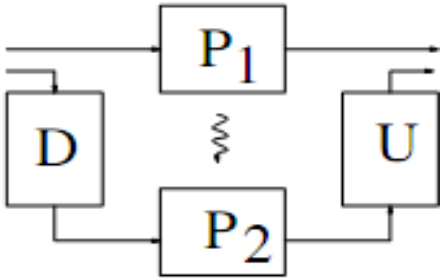
Under which **condition** security properties **preserved** ?

Security vs. Modularization: Solution



„**Rely-guarantee**“-formalization[1] of property.

Result: Get **conditions** for **compositionality**.



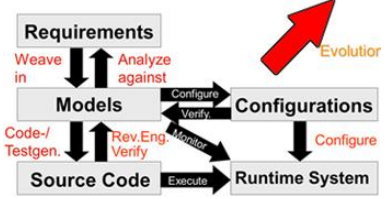
Theorem 5. *Let P_1, P_2, D and U be processes with $I_{P_1} = I_D$, $O_D = I_{P_2}$, $O_{P_2} = I_U$ and $O_U = O_{P_1}$ and such that D has a left inverse D' and U a right inverse U'. Let $m \in (\mathbf{Secret} \cup \mathbf{Keys}) \setminus \bigcup_{Q \in \{D', U'\}} (S_Q \cup K_Q)$.*
If P_1 preserves the secrecy of m and $P_1 \overset{(D,U)}{\rightsquigarrow} P_2$ then P_2 preserves the secrecy of m.

Proof: using formal semantics.

Above example: Rely-guarantee formalization shows that secure composition **impossible**.

[1] C.B. Jones 1981

Evolution-based Verification: Example

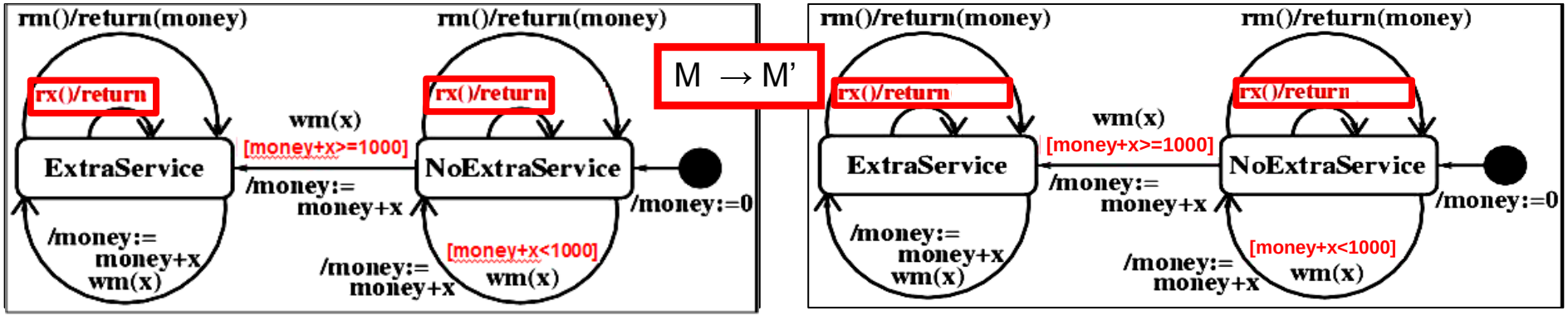


Secure information flow:

$$state_{current} \approx_{pub} state'_{current} \Rightarrow state_{current.t(m)} \approx_{pub} state'_{current.t(m)}$$

Evolution M → M‘: Only consider states for which:

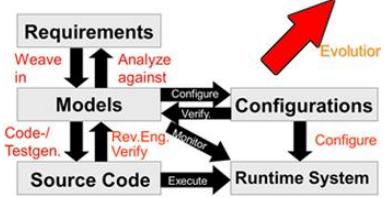
- $state_{current} \approx_{pub} state'_{current}$ in M‘, but not in M, or
- $state_{current.t(m)} \approx_{pub} state'_{current.t(m)}$ in M, but not in M‘.



Example: $wm(0).rx() \approx_{pub} wm(1000).rx()$ in M but not in M‘.

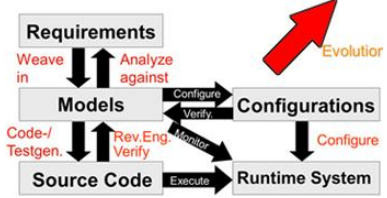
➔ M‘ **violates** secure information flow.

Secure Software Evolution



- Challenge and Scientific Basis ✓
- Secure Software Evolution
  - General Approach ✓
  - Preservation Results ✓
- Validation

Validation



- **Correctness**: Based on formal semantics.
- **Completeness**: Evolution: sequence of deletions / modifications / additions of system elements.

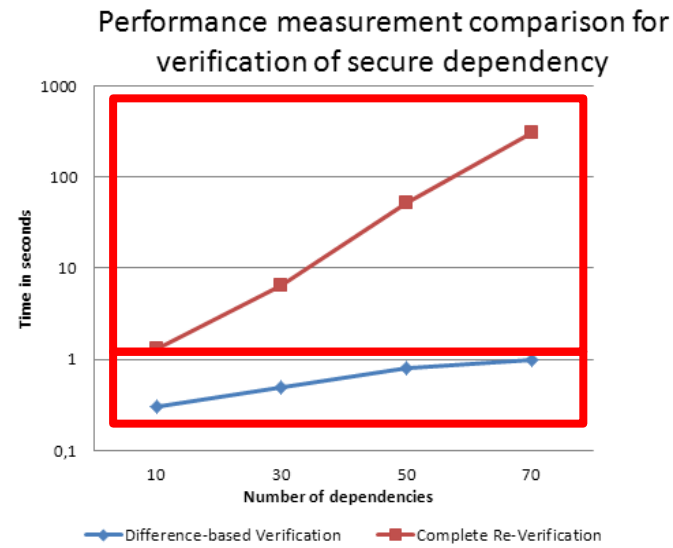
Performance gain **maximal** when **difference << software.** Example result:

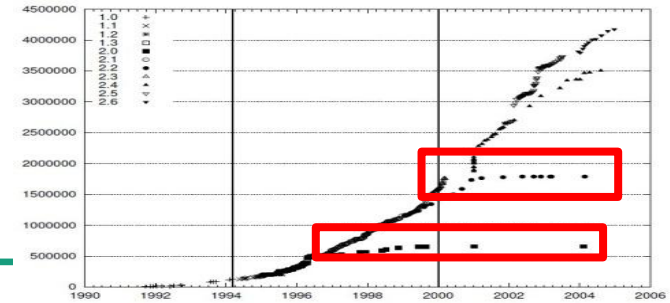
- Complete re-verification: Performance **exponential** in software size
- Evolution-based verification: Performance **linear** in software size

(each given **constant** size of **differences**).

Assumption satisfied e.g. for:

- **Maintenance** of stabile software versions
- Verification **integrated** with development





[Robles et al.: Evolution and Growth in Large Libre Software Projects]

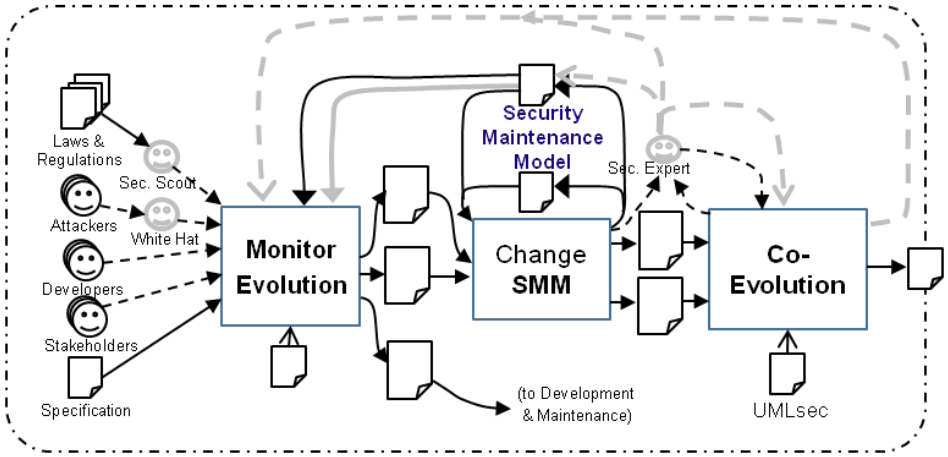
Current Project: Beyond One-Shot Security: Keeping Information Systems Secure through Environment-Driven Knowledge Evolution

Joint project with Stefan Gärtner, Kurt Schneider (Univ. Hannover) and Jens Bürger, Thomas Ruhroth, Johannes Zweihoff (TU Dortmund)

Goals

- Systematic co-evolution of knowledge and software for security.
- Develop techniques, tools, and processes
  - that support security requirements and design analysis techniques
  - for evolving, long-living systems
  - in order to ensure "lifelong" compliance to security requirements.

Approach



Some publications

- S. Gärtner, T. Ruhroth, J. Bürger, K. Schneider, J. Jürjens. Maintaining Requirements for Long-Living Software Systems by Incorporating Security Knowledge. In: 22nd IEEE International Requirements Engineering Conference (RE 2014), IEEE, 2014.
- J. Bürger and J. Jürjens and T. Ruhroth and S. Gärtner and K. Schneider. Model-based Security Engineering with UML: Managed Co-Evolution of Security Knowledge and Software Models. In A. Aldini, J. Lopez, F. Martinelli, Foundations of Security Analysis and Design VII: FOSAD 2012/2013 Tutorial Lectures, LNCS, vol. 8604, Springer, 2014, pp. 34–53.
- T. Ruhroth, J. Jürjens. Supporting Security Assurance in the Context of Evolution: Modular Modeling and Analysis with UMLsec. In 16th IEEE Intern. Symposium on High Assurance Systems Engineering (HASE 2012), IEEE, 2012, pp. 177–184.

Conclusion: Secure Software Evolution

Fraunhofer ISST

Integrate security verification with **evolution:**

- **Reuse** verification results by conditions for **security preservation** (e.g. **refinement**, **refactoring**, **patterns**, **modularization**).

➔ **Evolution-based verification**:

- Model-code **traceability** at evolution
- Evolution-based **static analysis** and **run-time verification**.

Validation: Successful application projects.

- Significant **performance gains**.

Current work: **Security certification** for **clouds**.

[Humberg, Wessel, Poggenpohl, Wenzel, Ruhroth, Jürjens. Using Ontologies to Analyze Compliance Requirements of Cloud-Based Processes. Cloud Computing and Services Science (selected best papers), LNCS Springer, 2014