Model-centric Security Verification Subject to Evolution

Jan Jürjens

TU Dortmund & Fraunhofer ISST

http://jan.jurjens.de

technische universität
dortmund

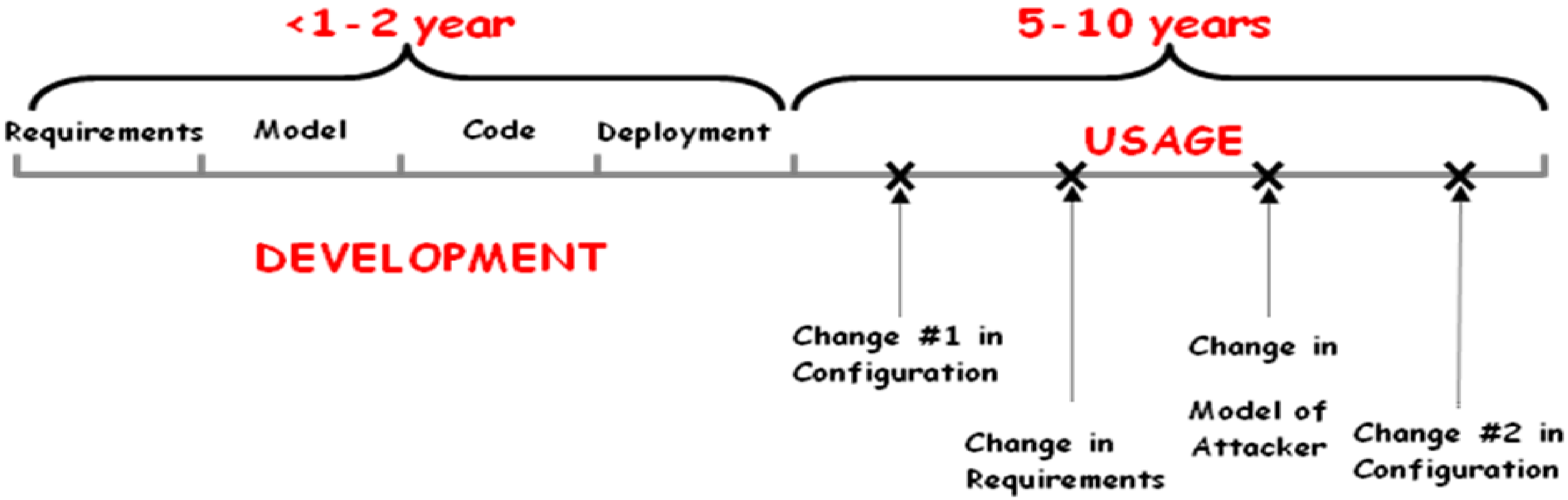
Fraunhofer
ISST

The Forgotten End of the System Life-cycle

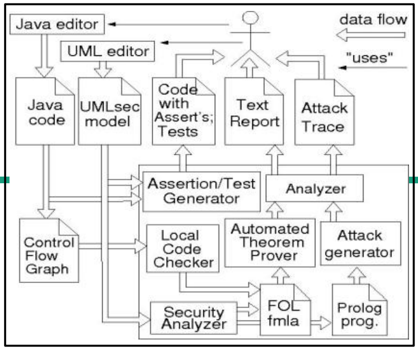
Challenges:

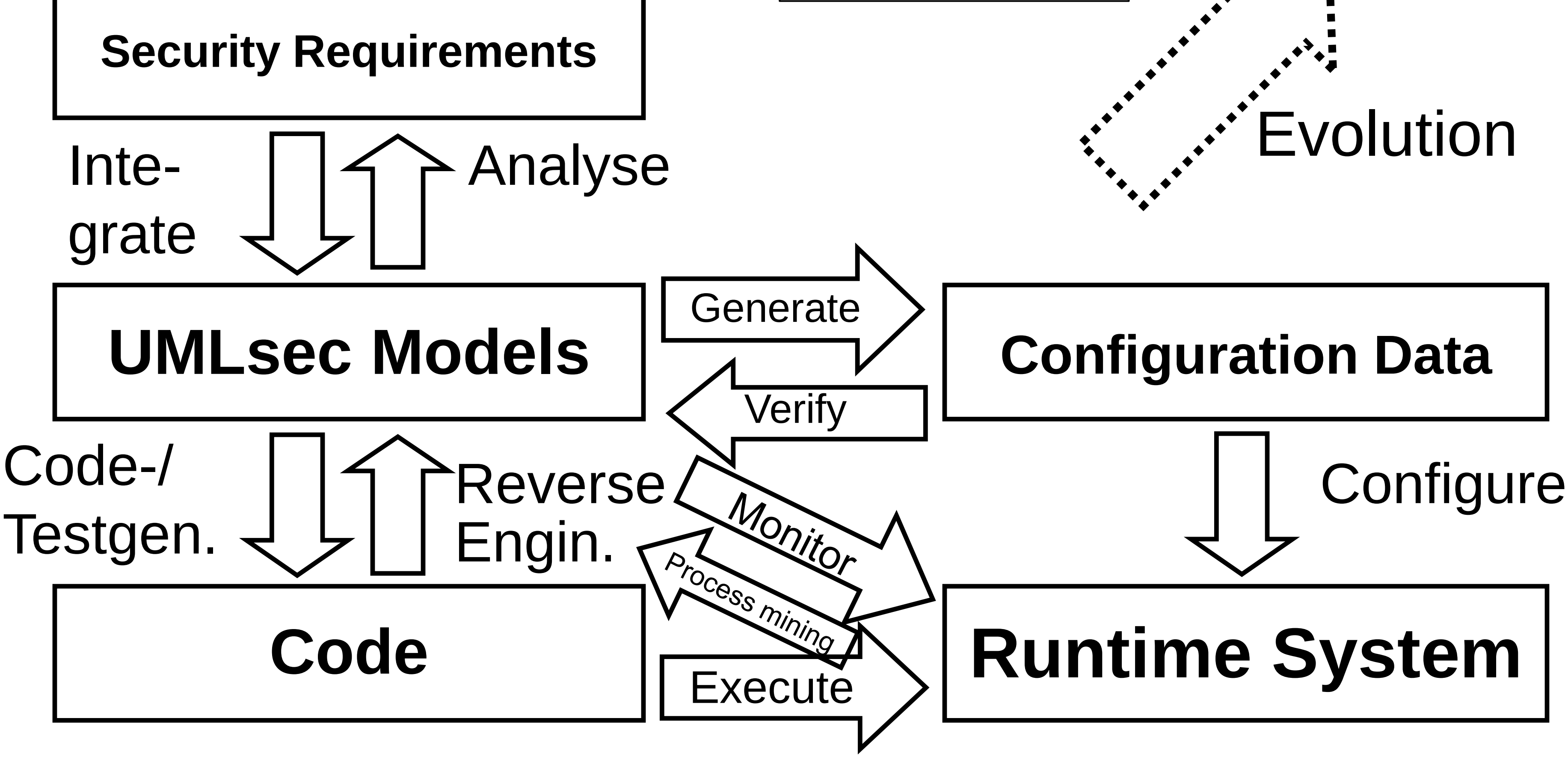
- Software lifetime often longer than intended (cf. Year-2000-Bug).
- Systems evolve during their lifetime.
- In practice evolution is difficult to handle.

Problem: Critical requirements (e.g. security) preserved ?



Model-based Security Engineering with UMLsec





Security Requirements

Inte-grate

Analyse

Evolution

UMLsec Models

Generate

Verify

Configuration Data

Code-/Testgen.

Reverse Engin.

Monitor

Process mining

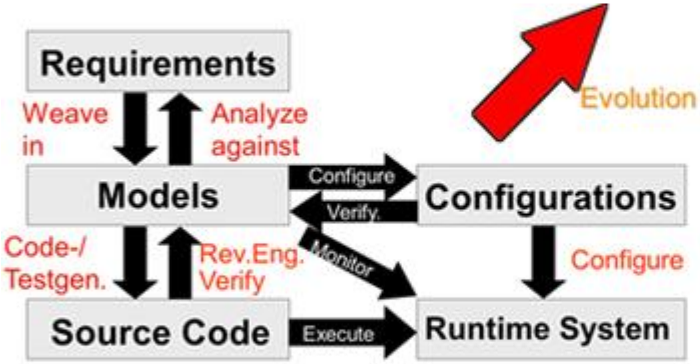
Configure

Code

Execute

Runtime System

Challenge: Evolution



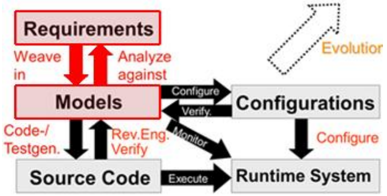
Each artifact may evolve.

To reduce costs, reuse verification results as far as possible.

⇒ Under which conditions does evolution preserve security?

Even better: examine possible future evolution for effects on security.

- Check *beforehand* whether potential evolution will preserve security.
- Choose an architecture during the design phase which will support future evolution best wrt. security.



Model Formalization

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

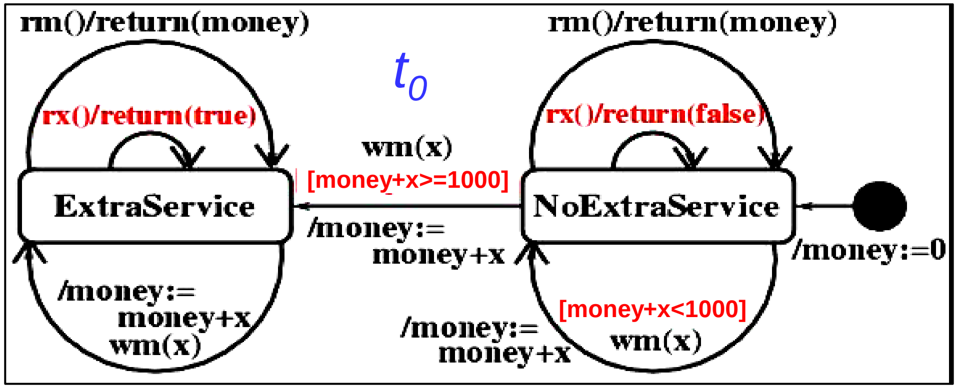
$$Exec(t,m) = [state_{current}=source \wedge m=msg \wedge cond[m]=true \Rightarrow action[m] \wedge state_{current.t(m)}=target].$$

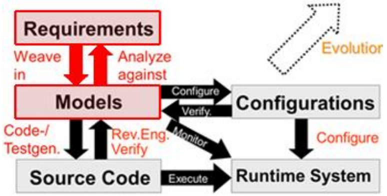
(where $state_{current}$ current state; $state_{current.t(m)}$ state after executing t).

Example: Transition t_0:

[Jürjens, Fox: Tools for Model-based Security Engineering. ICSE'06]

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





Formalization of Requirements

Example „secure information flow“:
No information flow from confidential to public data.

Analysis: If two states $state_{current}$, $state'_{current}$ differ only in confidential attributes, then their publically observable behaviour needs to be the same:

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

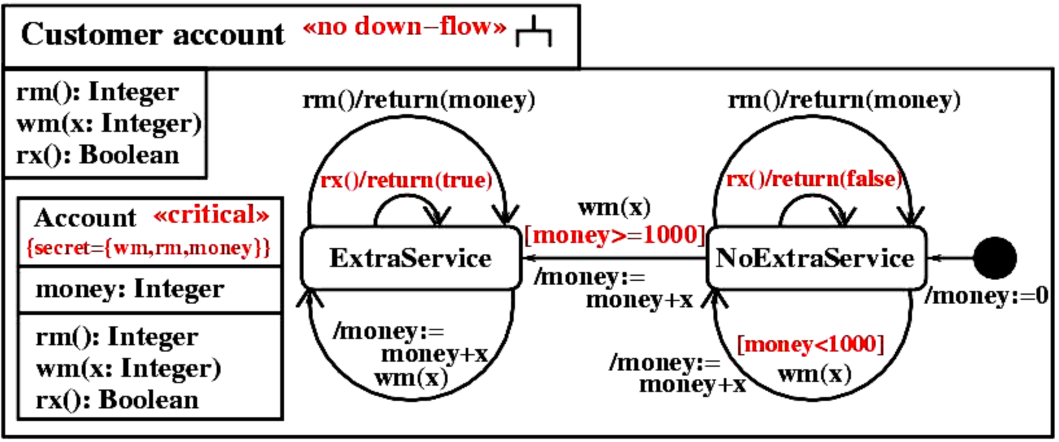
(where $state_{current} \approx_{pub} state'_{current}$ if $state_{current}$ and $state'_{current}$ have the same publically observable behaviour).

Example: Insecure, because confidential attribute *money* influences return value of public method *rx()*.

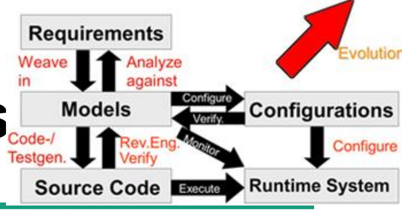
ExtraService $\approx_{pub}$ *NoExtraService*

aber nicht:

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



Evolution vs. Design- / Architectural Principles



Consider design techniques and architectural principles which support evolution.

Under which conditions are requirements preserved ?

Design technique: **Refinement of specifications.** Supports evolution between refinements of an abstract specification.[1]

Architectural principle: **Modularization** supports evolution by restricting impact of change to modules. Different dimensions:

- **Architectural layers**
- **Component**-oriented architectures
- **Service**-oriented architectures
- **Aspect**-oriented architectures

For each discovered conditions under which requirements are preserved. Explain this at the hand of security requirements.

[1] [Schmidt, Jürjens: Connecting Security Requirements Analysis and Secure Design Using Patterns and UMLsec. CAiSE'11]

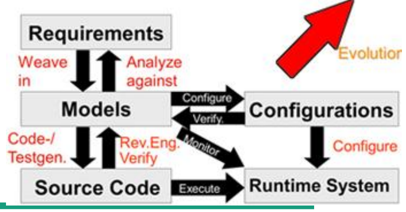
[Hatebur, Heisel, Jürjens, Schmidt: Systematic Development of UMLsec Design Models Based on Security Requirements. FASE'11]

[Ochoa, Jürjens, Warzecha: A Sound Decision Procedure for the Compositionality of Secrecy. ESSoS'12]

[Deubler, Grünbauer, Jürjens, Wimmel: Sound development of secure service-based systems. ICSOC'04]

[Jürjens, Houmb: Dynamic Secure Aspect Modeling with UML. MoDELS'05]

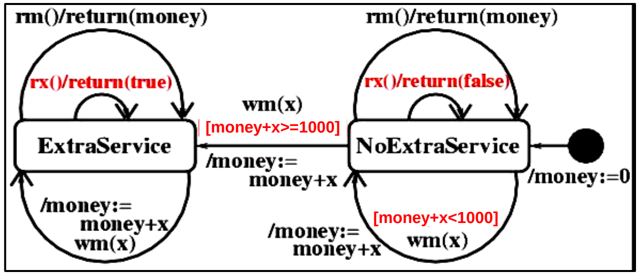
Design Technique: Refinement



For behaviour preserving refinement, one would expect preservation of behavioural requirements.

„**Refinement Paradox**“: Surprisingly, in general not true [Roscoe‘96].

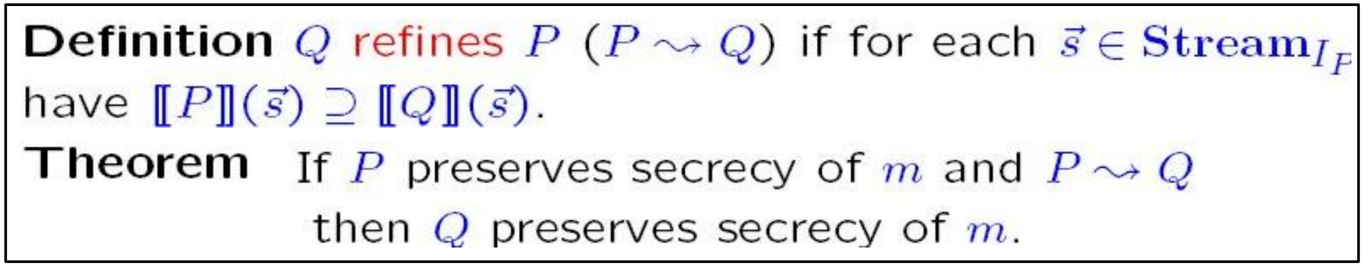
Example: In above example, transition *rx()/return(true)* (resp. *false*) is refinement of „secure “ transition *rx()/return(random_bool)*.



Observation: Problem: Mixing non-determinism as under-specification resp. as security mechanism. Our specification approach separates these.

Result: Refinement now preserves behavioural requirements.

Proof: using formal semantics.

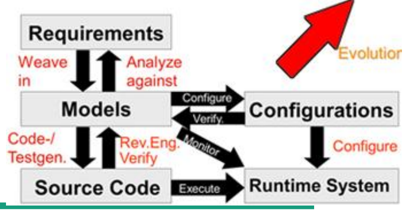


Definition Q refines P ($P \leadsto Q$) if for each $\vec{s} \in \mathbf{Stream}_{I_P}$ have $[\![P]\!](\vec{s}) \supseteq [\![Q]\!](\vec{s})$.

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

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

Architectural Principle: Modularization



Problem: Behavioural requirements in general not compositional.

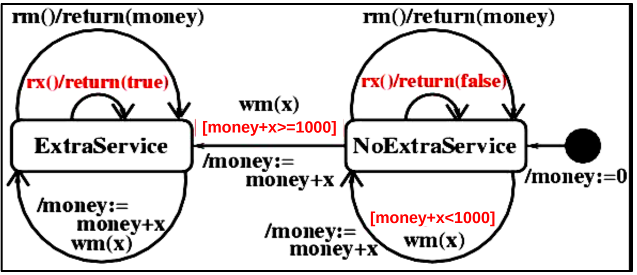
Above example: States *ExtraService* and *NoExtraService* each „secure " (only one return value for *rx*), but composition in statechart not.

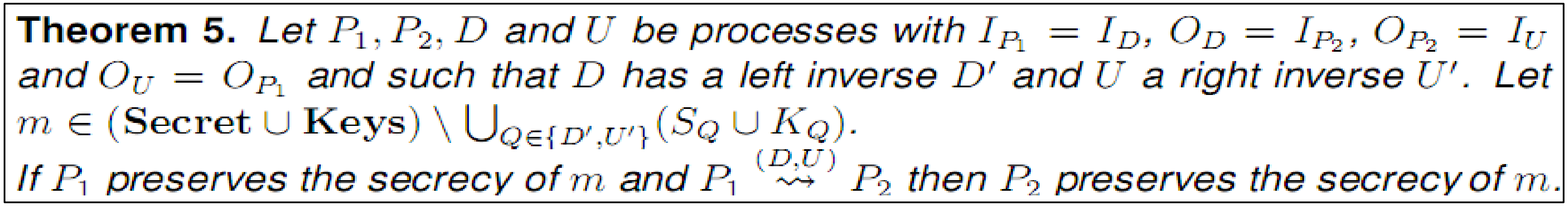
Under which condition are requirements preserved ?

Solution: Formalize requirement as „rely-guarantee"-property.

Result: Using this formalization, get conditions for compositionality.

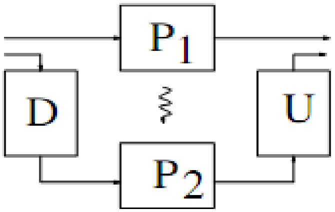
Proof: using formal semantics.



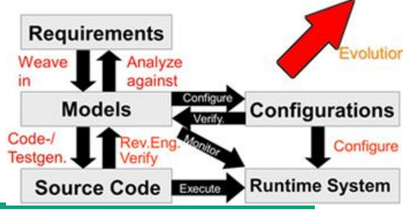


Theorem 5. *Let* P_1, P_2, D *and* U *be processes with* $I_{P_1} = I_D$, $O_D = I_{P_2}$, $O_{P_2} = I_U$ *and* $O_U = O_{P_1}$ *and such that* D *has a left inverse* D' *and* U *a right inverse* U'. *Let* $m \in (\mathbf{Secret} \cup \mathbf{Keys}) \setminus \bigcup_{Q \in \{D',U'\}} (S_Q \cup K_Q)$.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

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



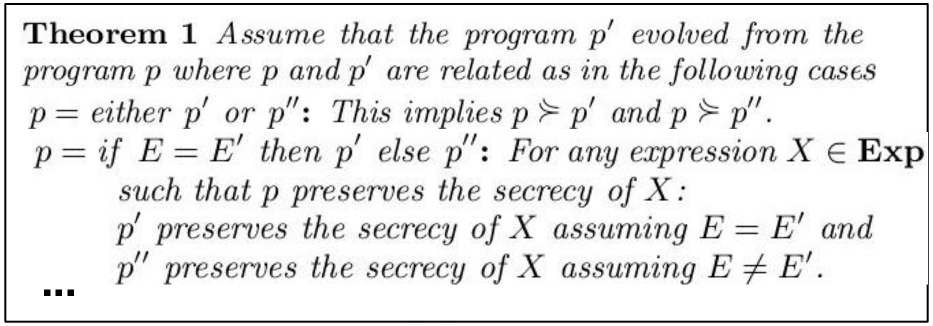
Evolution-based Verification



Evolution-based Verification – Idea:

- Initial verification: Tool registers which **model elements** relevant for verification of given requirement.
- Store in verified model, together with partial results („proof-carrying models").
- Discovered **conditions on changes** such that requirement preserved.
- **Compute difference** between old and new model (e.g. using SiDiff [Kelter]).
- Only need to re-verify **model parts** which
  1) have **changed**
  2) were **relevant** in the initial verification and
  3) which don't satisfy the above-mentioned conditions.

Significant verification speed-up compared to simple re-verification.

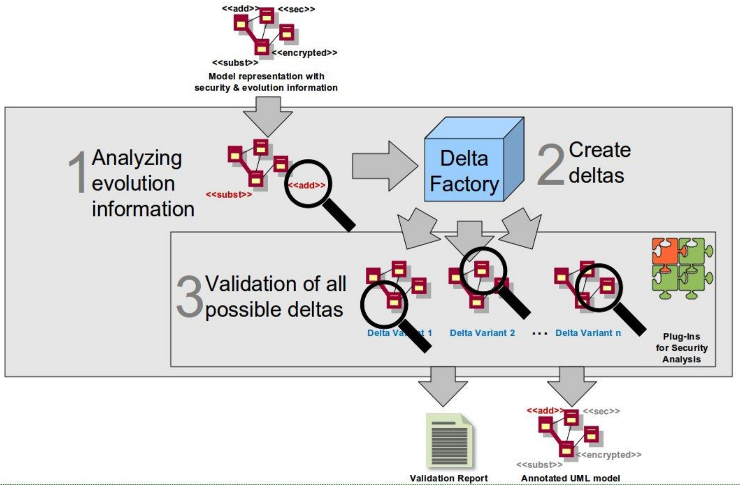


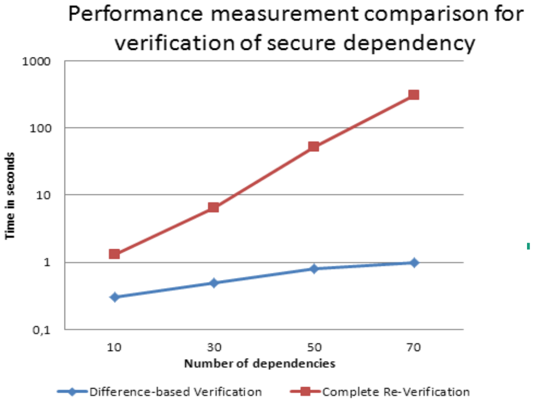
Theorem 1 *Assume that the program* p' *evolved from the program* p *where* p *and* p' *are related as in the following cases*

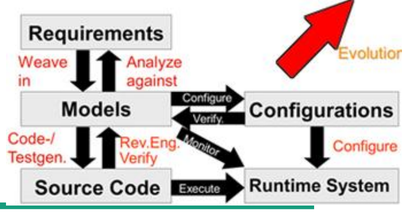
$p =$ *either* p' *or* p'': *This implies* $p \succcurlyeq p'$ *and* $p \succcurlyeq p''$.

$p =$ *if* $E = E'$ *then* p' *else* p'': *For any expression* $X \in \mathbf{Exp}$
such that p *preserves the secrecy of* X:
p' *preserves the secrecy of* X *assuming* $E = E'$ *and*
p'' *preserves the secrecy of* X *assuming* $E \neq E'$.

...



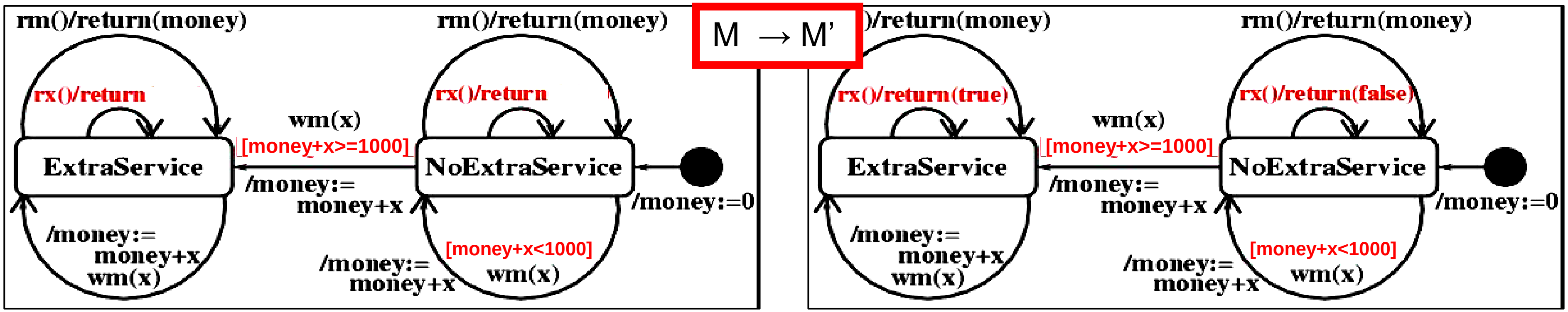




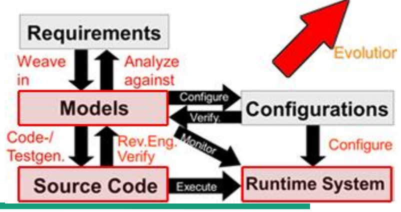
Evolution-based Verification: Example

Preservation condition for secure information flow at evolution M → M‘: Only consider states s, s' for which:

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



Example: $wm(0).rx() \approx_{pub} wm(1000).rx()$ in M but not in M‘. Shows that M‘ violates secure information flow (confidential data *0* and *1000* distinguishable).



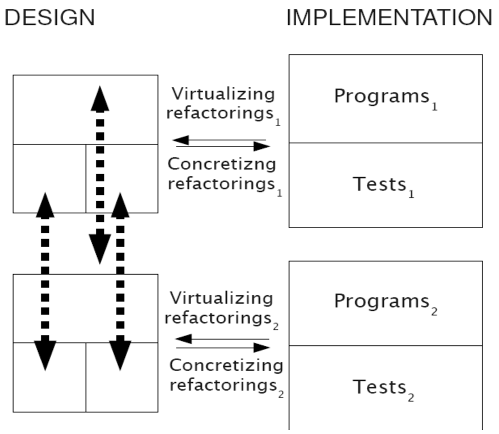
Model-code Traceability under Evolution

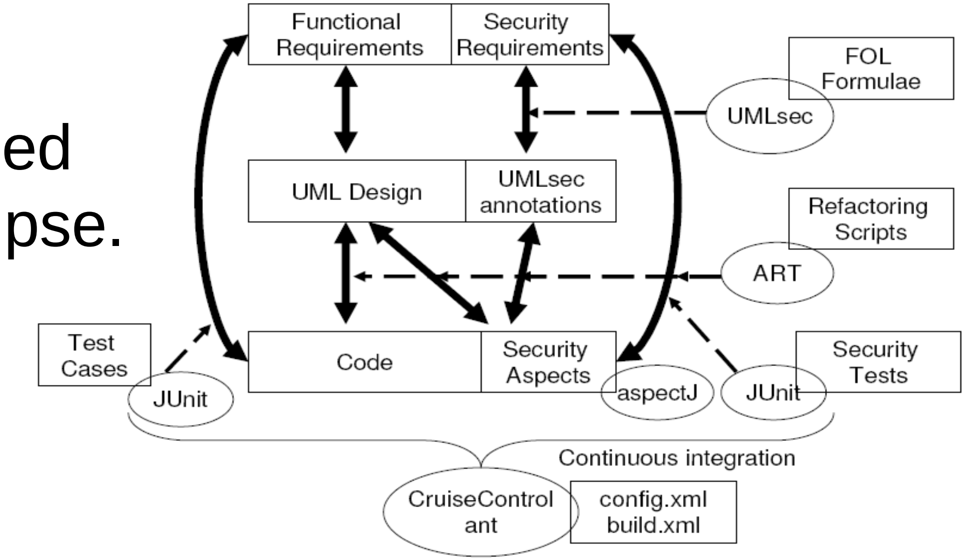
Goal: Preserve model-code traceability during evolution.

Idea: Reduce evolution to:

- Adding / deleting model elements.
- Supporting refactoring operations.

=> Approach for automated model-code traceability based on refactoring scripts in Eclipse.





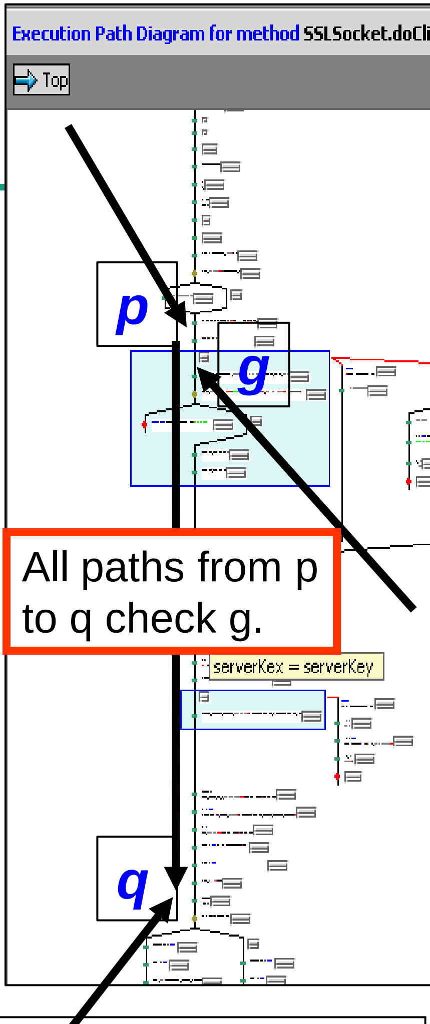
[Bauer, Jürjens, Yu: Run-Time Security Traceability for Evolving Systems. Computer Journal '11]

Code Verification subject to Evolution

Use evolution-based model verification and model-code traceability for evolution-aware code verification using static analysis.

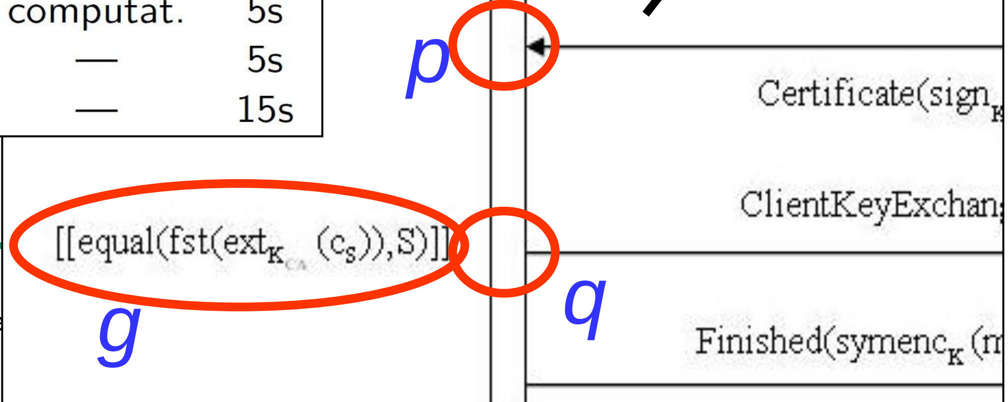
Example: Condition in sequence diagram correctly checked in implementation.

Project Csec (with Microsoft Research Cambridge): Implemented static analysis, found several weaknesses.

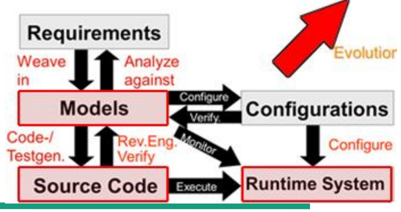


All paths from p to q check g.

	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



[Jürjens. Security Analysis of Crypto-based Java Programs using Automated Theorem Provers. ASE'06.]
[Aizatulin, Gordon, Jürjens: Extracting and verifying cryptographic models from C protocol code by symbolic execution. CCS'11]



Run-time Verification subject to Evolution

Relevant versions of source code not always available => run-time monitoring.

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

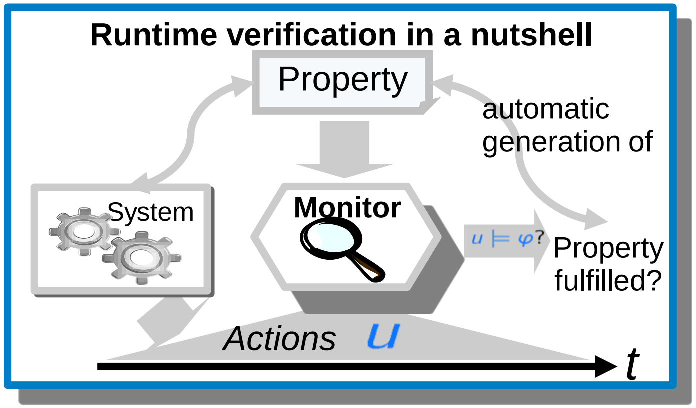
Problem: no evolution and only **„safety“-properties** supported (too restrictive e.g. for secure information flow).

So: New approach, based on runtime verification (based on techniques from model-checking and testing).

Formalize requirement to be monitored in LTL.

Continuous monitoring of system events through monitors generated from the models, **with evolution-based traceability.**

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


Runtime verification in a nutshell

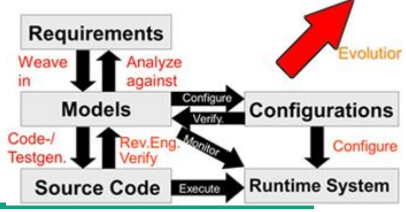
Example results:

[Bauer, Jürjens. Runtime Verification of Cryptographic Protocols. Computers & Security ‘10]

[Pironti, Jürjens. Formally-Based Black-Box Monitoring of Security Protocols. ESSOS’10]

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

Technical Validation

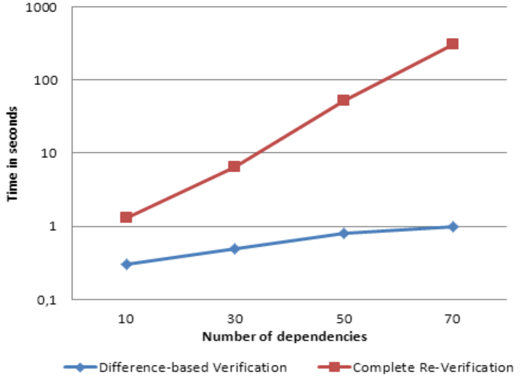


- **Correctness**: based on formal semantics. ✓
- **Completeness**: view model transformation as sequence of deletions, modifications and additions of model elements. ✓

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

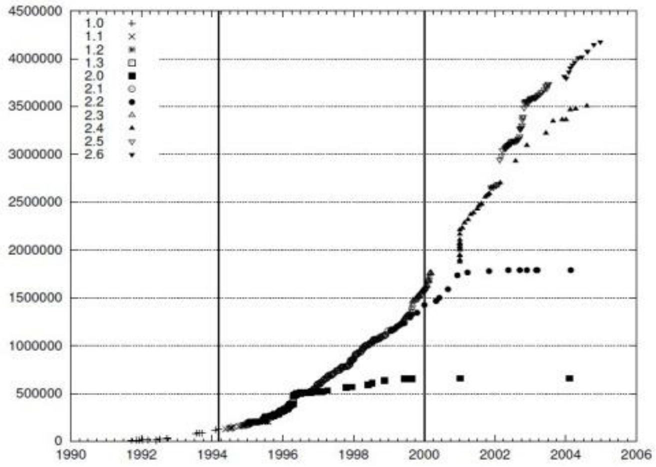
- Evolution-based verification:
  Performance **linear** in software size
  (given constant size of differences)
- Complete Re-Verification:
  Performance **exponential** in software size.

Performance measurement comparison for verification of secure dependency



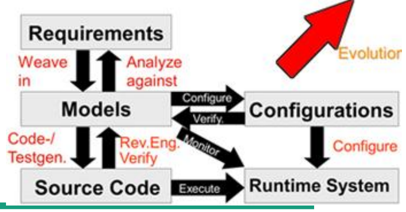
This condition is satisfied e.g. for:

- **Maintenance of stabile software**
- **QA tightly integrated with evolution** (e.g. nightly builds)



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

Practical Validation



Application of in practice (examples):

- Global Platform (smartcard software updates, Gemalto)
- Mobile software architecture (Telefonica O2 Germany)
- Internal information system (BMW)
- Biometric authentication system
- German Health Card
- Health information systems

[Jürjens et al.: Incremental Security Verification for Evolving UMLsec models. ECMFA'11]

[Jürjens et al.: Model-based Security Analysis for Mobile Communi-cations. ICSE'08]

[Best, Jürjens, Nuseibeh: Model-based Security Engineering of Distributed Information Systems using UMLsec, ICSE'07]

[Lloyd, J. Jürjens, Security Analysis of a Biometric Authentication System using UMLsec and JML. Models'09]

[Jürjens, Rumm: Model-based Security Analysis of the German Health Card Architecture. Methods of Information in Medicine'08]

[Mouratidis, Sunyaev, Jürjens: Secure Information Systems Engineering: Experiences and Lessons Learned from Two Health Care Projects. CAiSE'09]

Detected signification weaknesses for some of these.

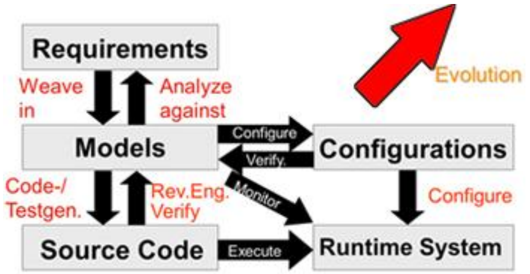
Empirical comparison model-based vs. traditional QA (testing):

Example: **Model-checking vs. simulation / testen**:

Door control unit (coop. w. BMW). Model-checking: Additional effort (1-2 days / LTL formula), but detects also obscure bugs.

[Jürjens, Trachtenherz, Reiss: Model-based Quality Assurance of Automotive Software. Models'08]

Conclusion: Model-centric Security Verification Subject to Evolution



Evolution: challenging for QA.

Question: Can reuse QA results after evolution ?

Result: Condition for requirements preservation…

- … in context of design-/architectural techniques for evolution (e.g. **refinement**, **modularization**).
- … under model evolution („**evolution-based verification**“).
- evolution-based **static analysis** and **run-time verification**.
- Tool-implementation: significant **performance** and scability **gains** wrt. simple re-verification.

Validation: Successful use in practice.