Vorlesung

Methodische Grundlagen des Software-Engineering

im Sommersemester 2014

Prof. Dr. Jan Jürjens

TU Dortmund, Fakultät Informatik, Lehrstuhl XIV

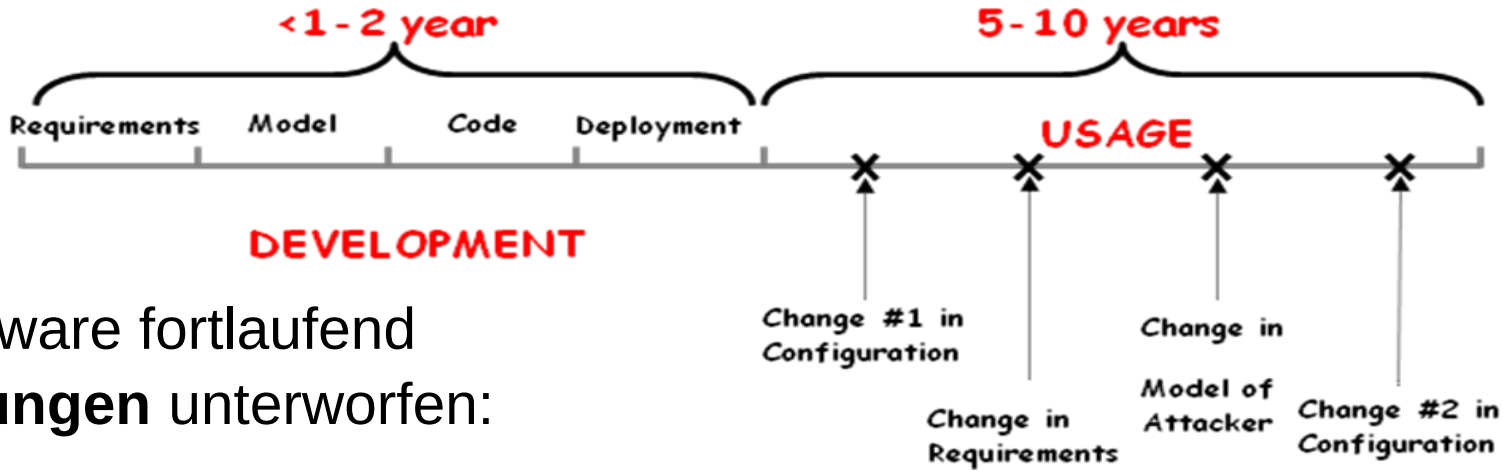
Teil 3.4a: Sichere Software-Evolution

v. 08.07.2014

Was ist Softwareevolution ?

Software heute oft sehr „**langlebig**“ (vgl. Jahr-2000-Fehler).

„Nutzt“ sich nicht ab.



Aber: Software fortlaufend **Änderungen** unterworfen:

- Änderungen der **Anforderungen**
- Änderungen der **Softwareumgebung**
- **Fehlerkorrektur**

➔ „**Softwareevolution**“

Was ist dabei die Herausforderung ?

Nach Änderung Software neu testen (**Regressionstest**):

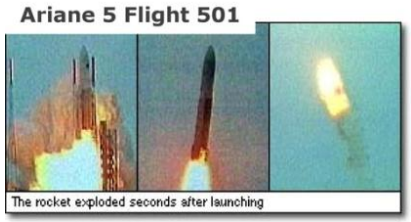
Alle Tests wiederholen (auch wenn nur Umgebung geändert).

Hohe Kosten:

- z.B. **Jahr-2000**-Problem:
  Weltweit ca. 600 Mrd US$ Kosten.

➔ Oft **unzureichende Regressionstests**:

- Selbstzerstörung der **Ariane 5** (1996):
  Ariane 4-Software in Ariane 5 wiederverwendet.
  ➔ 370 Mio US$ Schaden.



Welche Softwareengineering-Techniken können helfen?

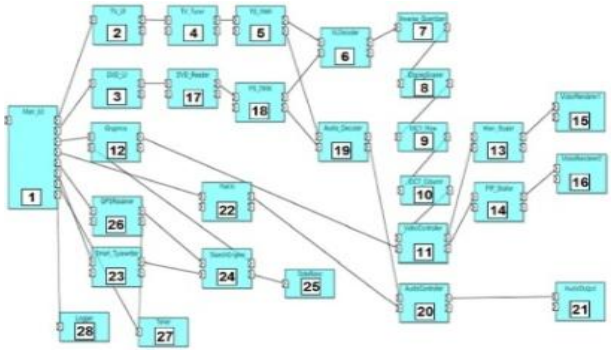
Hilfreich: Designtechniken und Architekturprinzipien, die Management von Evolution unterstützen.

Designtechnik: **Verfeinerung von Spezifikationen.**

➔ Evolution zwischen Verfeinerungen einer Spezifikation.

Architekturprinzip Modularisierung

➔ Auswirkungen auf Systemteile beschränken.



Problem: Verfeinerung & Modularisierung bewahren **nicht alle** Anforderungen (z.B. Sicherheitsanforderungen) !

Wie können wir dieses Problem lösen ?

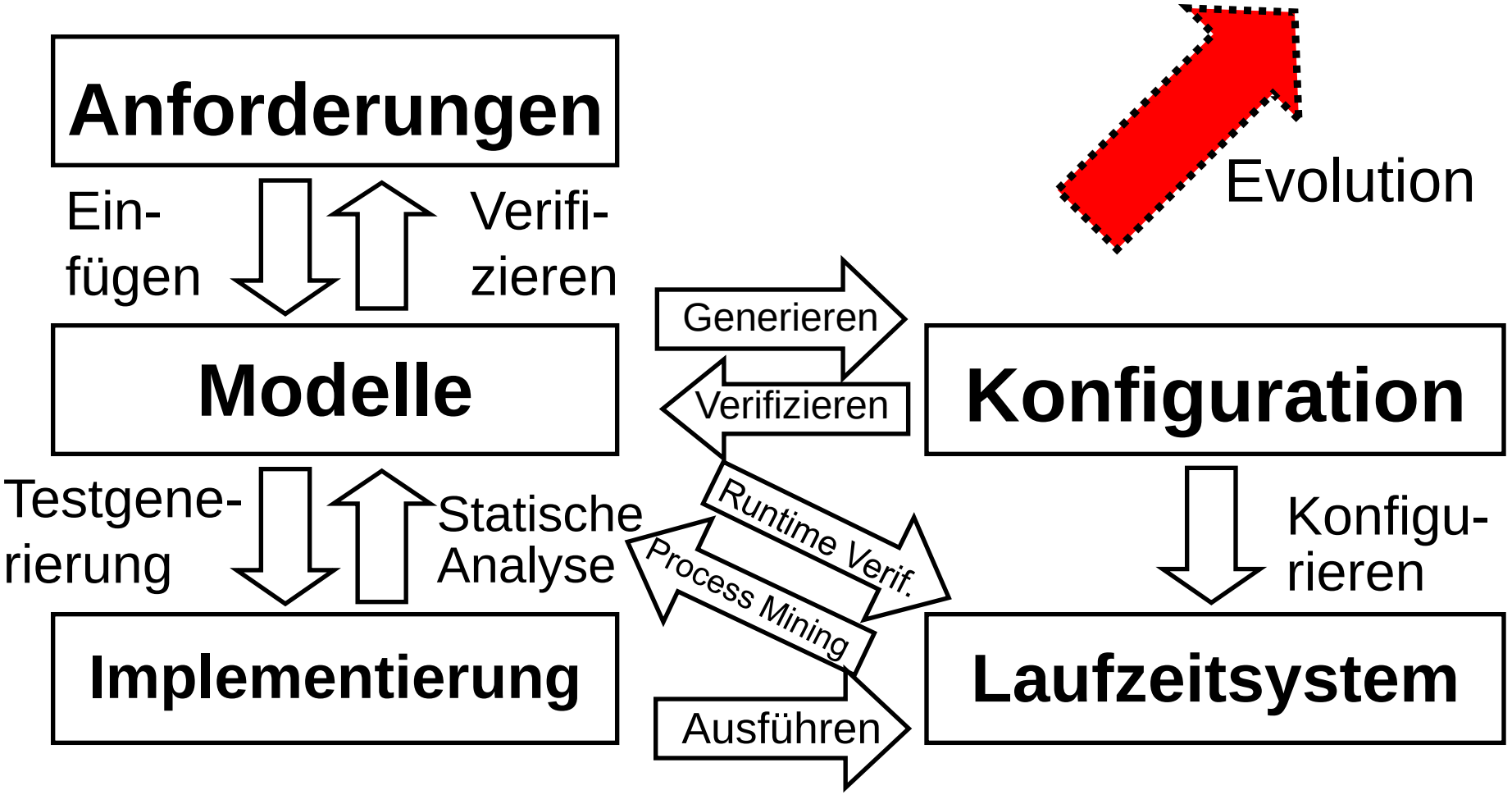
Ziel: Qualitätssicherung, die **Evolution** unterstützt:

- QS-Resultate soweit möglich **wiederverwenden**.

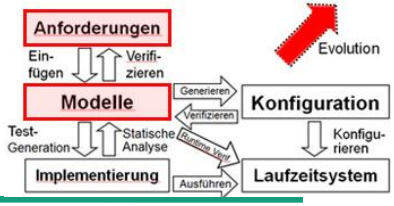
➔ Unter welchen **Bedingungen Anforderungen bewahrt** ?

- Erneute Überprüfung nur, wenn Anforderungen **nicht** bewahrt.
- Nur Software-Teile erneut überprüfen, für die **notwendig**.
- Verschiedene **Evolutions-Alternativen** automatisch auf Auswirkungen auf Anforderungen analysieren.
- Im Voraus: **Architektur** wählen, die bei Evolution Bewahrung kritischer Anforderungen optimal unterstützt.

Grundlage: Modellbasierte Qualitätssicherung



Evolution vs. Design- / Architekturprinzipien



Unter welchen Voraussetzungen bewahren Verfeinerung und Modularisierung Anforderungen ?

Verfeinerung: Verfeinerungsansatz entwickelt, der Sicherheitsanforderungen **bewahrt**.

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

Modularisierung: z.B.:

- Schichtung von **Architekturebenen**
- **Komponenten**-orientierte Architekturen

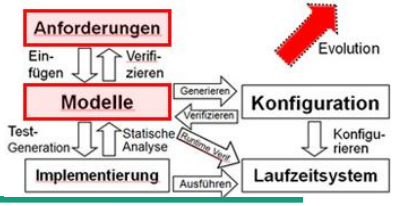
[Hatebur, Heisel, Jürjens, Schmidt: Systematic Development of UMLsec Design Models Based on Security Requirements. FASE'11]
[Ruhroth, Jürjens: Supporting Security Assurance in the Context of Evolution: Modular Modeling and Analysis with UMLsec. HASE'12]

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

Bedingungen für Bewahrung von Anforderungen identifiziert.

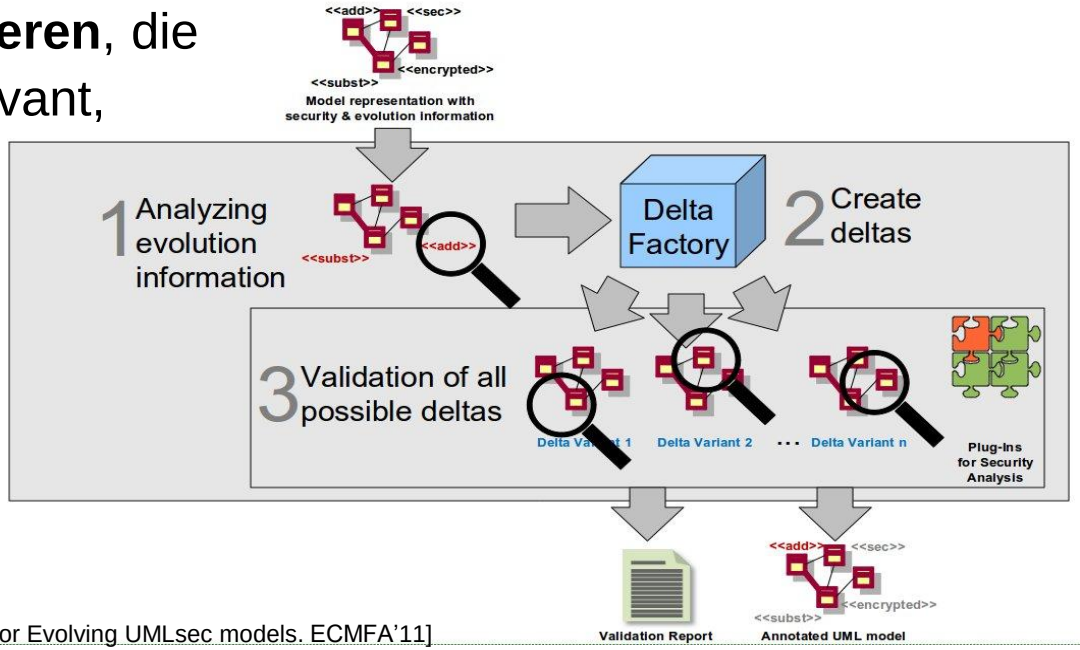
Im Folgenden am Beispiel von Sicherheitsanforderungen.

Evolutions-basierte Verifikation



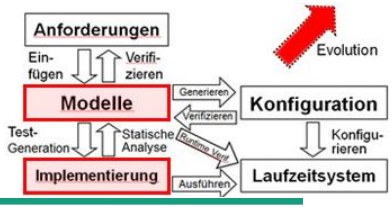
- Erstmalige Analyse: Registrieren, welche **Modellelemente relevant**.
- Teilresultate in Modell speichern („**proof-carrying models**“).
- **Differenz**: altes zu neues Modell berechnen (z.B. mit SiDiff [Kelter]).
- Obige Resultate für **sicherheitsbewahrende Änderungen** verwenden.
- Nur die **Modellteile reverifizieren**, die
  1) in der initialen Analyse relevant,
  2) **geändert** wurden, sodass
  3) o.g. Bedingungen **nicht erfüllt**.

➔ Erheblich **weniger Aufwand** als komplette Reverifikation.



[Jürjens, Marchal, Ochoa, Schmidt: Incremental Security Verification for Evolving UMLsec models. ECMFA'11]

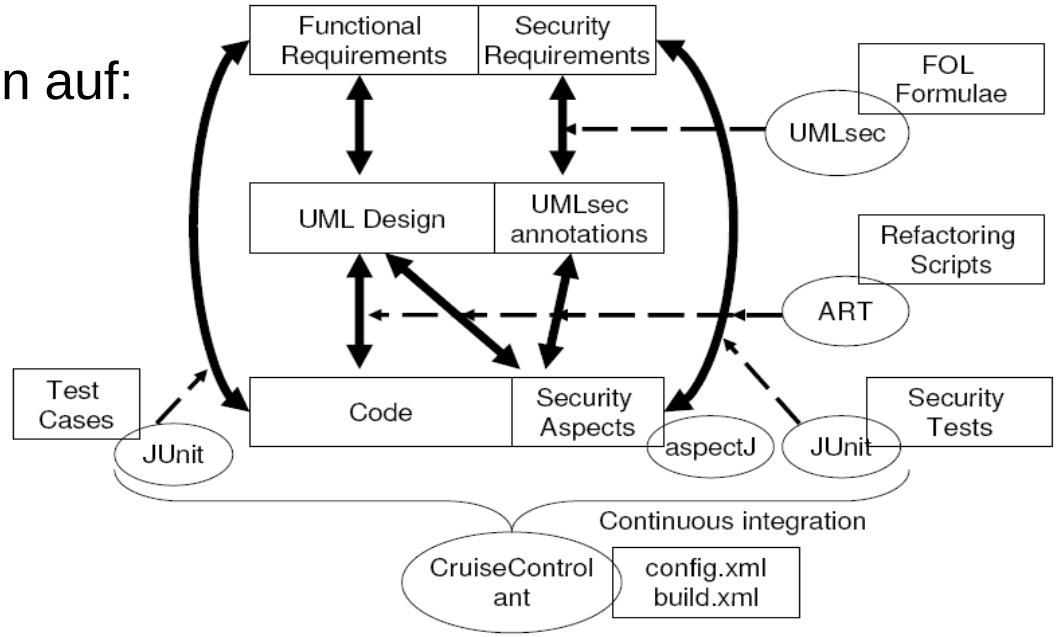
Verbindung von Modell und Implementierung bei Evolution



Ziel: Nachverfolgbarkeit von **Anforderungen** vs. **Implementierung** bei **Evolution**.

Lösung: Änderungen reduzieren auf:

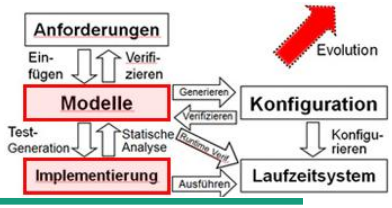
- Hinzufügen / Entfernen von Systemteilen.
- Grundlegende Refactoring-Operationen.



➔ **Automatische Nachverfolgbarkeit** der Änderungen zwischen Modell und Implementierung mit **Refactoring Scripts** (Eclipse).

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

Sicherheitsanalyse auf Implementierungs-Ebene



Sicherheitsschwachstelle 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/März 2014: "**goto fail**"-Schwachstellen in SSL @ iPhone, GnuTLS

Sicherheitsanalyse mittels symbolischer Ausführung

Übersetzung von C in symbolisches Modell für Sicherheitsanalyse.

Beispiel-Nachricht: $A \xrightarrow{m, hash(m, k_{shared})} B$

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
}
```

Modell nach **symbolischer Ausführung**:

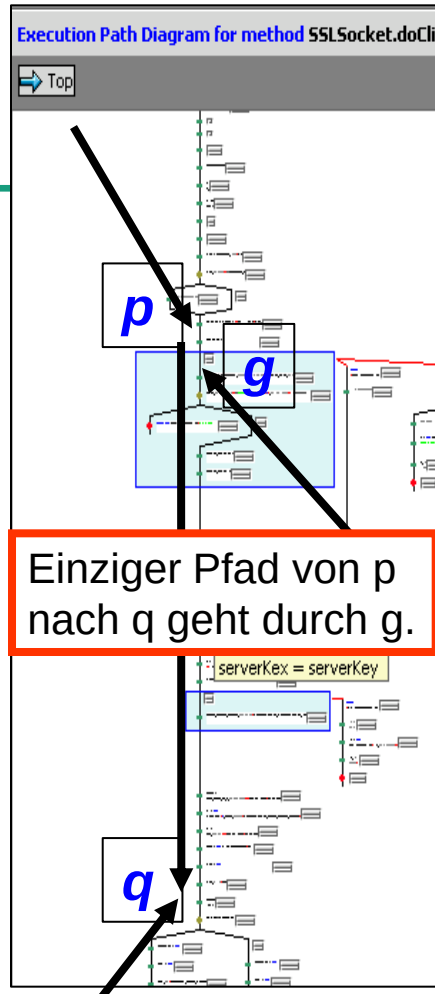
$$\mathbf{out}(\mathbf{len}(\text{payload}) \mid 01 \mid \text{payload} \mid \text{sha1}(\mathbf{len}(\text{payload}) \mid 01 \mid \text{payload}, k_{shared}))$$

Evolutions-basierte Verifikation auf Implementierungs-Ebene

Automatische statische Analyse der Implementierung gegenüber Modell (z.B. Bedingungen in Sequenz-diagramm korrekt in Implementierung umgesetzt).

Projekt Csec mit Microsoft Research Cambridge: Werkzeug erfolgreich eingesetzt, mehrere Schwachstellen gefunden.

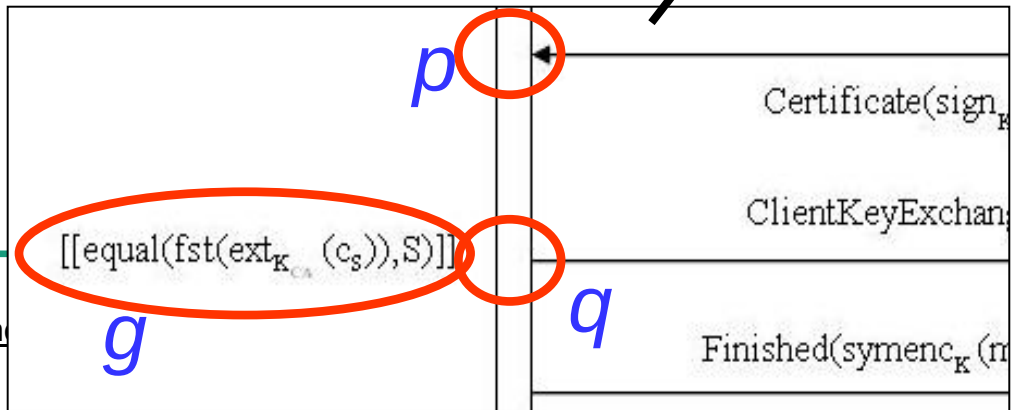
Evolutions-basierte Verifikation mittels Nachverfolgbarkeit.



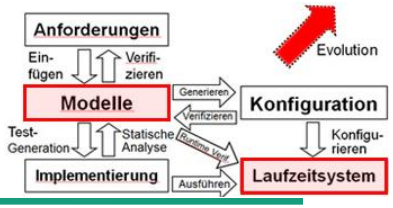
Einziger Pfad von p nach q geht durch g.

	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

[Aizatulin, Gordon, Jürjens: Extracting and verifying cryptographic models from C protocol code by symbolic execution. CCS'11]
[Aizatulin, Gordon, Jürjens : Computational Verification of C Protocol Implementations by Symbolic Execution. CCS'12]
[Dupressoir, Gordon, Jürjens, Naumann: Guiding a General-Purpose C Verifier to Prove Cryptographic Protocols. Journal of Computer Security 2014]



Evolutions-basierte Laufzeitverifikation



Quellcode nicht verfügbar ➔ Laufzeitüberwachung.

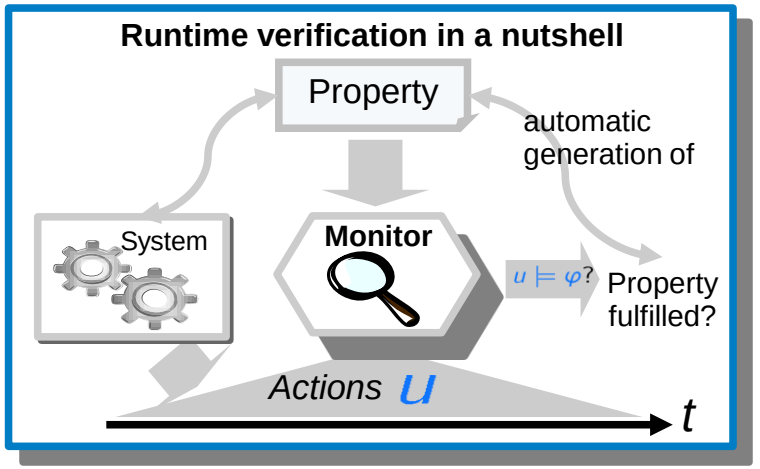
Möglicher Ansatz für Monitoring: Security Automata [F.B. Schneider 2000].

Problem: keine Evolution; nur **„Safety“-Eigenschaften**.

Neuer Ansatz auf Basis von Runtime-Verification[1].

- Sicherheitsanforderung in LTL.
- Generierung von Monitoren; **automatische Evolutions-Anpassung.**

Auch **Nicht-Safety-Eigenschaften** (3-wertige LTL-Semantik).

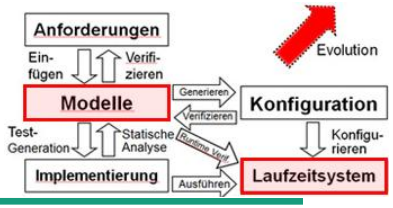


[Bauer, Jürjens. Runtime Verification of Cryptographic Protocols. Computers & Security ‘10]
[Pironti, Jürjens. Formally-Based Black-Box Monitoring of Security Protocols. ESSOS’10]

[1] Havelund, Grosu 2002

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

Für Monitoring: 3-wertige LTL-Semantik



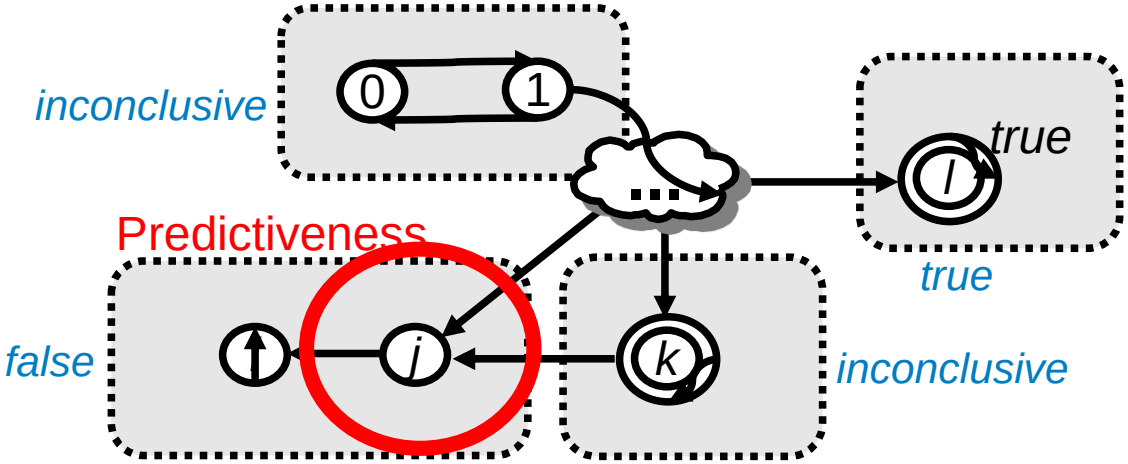
Ziel beim Monitoring: So früh wie möglich feststellen, dass Anforderung verletzt werden wird.

Verwende **3-wertige Semantik**.

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

➔ Endlicher Zustands-automat, um minimale Präfixe für unsicheren Zustand zu finden.

Auch für **Nicht-Safety-Eigenschaften**.



Beispiel-Anforderung: Server Finished

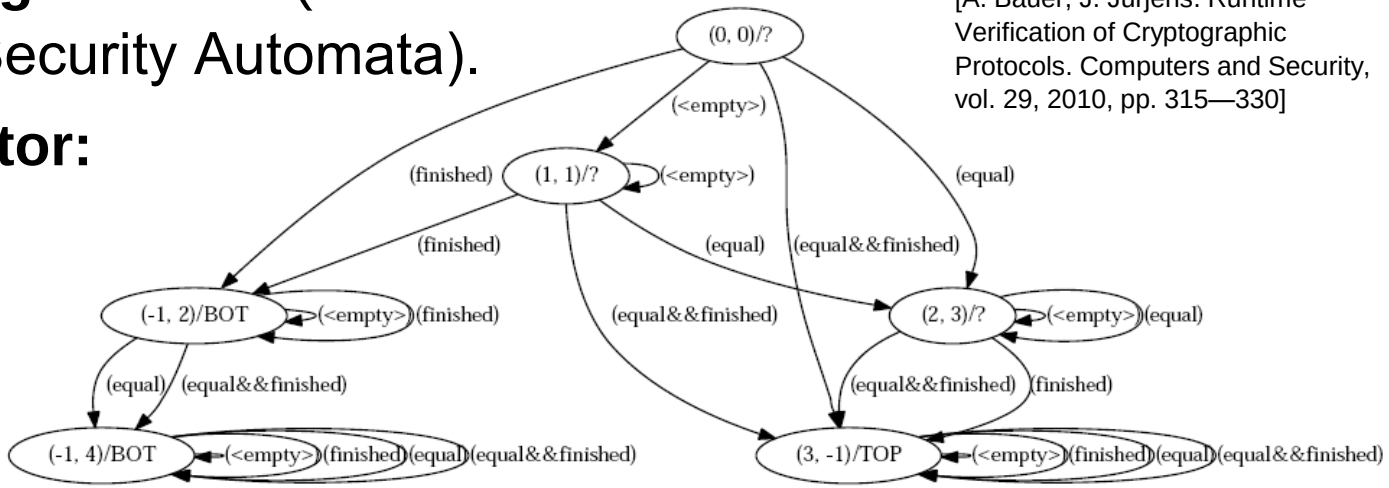
Server schickt Nachricht Finished **nicht** (Ereignis “finished”), **bevor** Nachricht Finished vom Client **erhalten** und enthaltener MD5-Wert **gleich** MD5 auf Seiten des Servers (Ereignis “equal”). **Dann** wird sie herausgeschickt. Formalisiert in LTL:

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

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

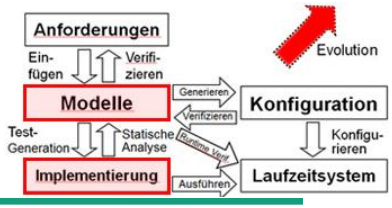
Keine Safety-Eigenschaft (d.h. nicht mit Schneider’s Security Automata).

Erzeugter Monitor:



[A. Bauer, J. Jürjens. Runtime Verification of Cryptographic Protocols. Computers and Security, vol. 29, 2010, pp. 315—330]

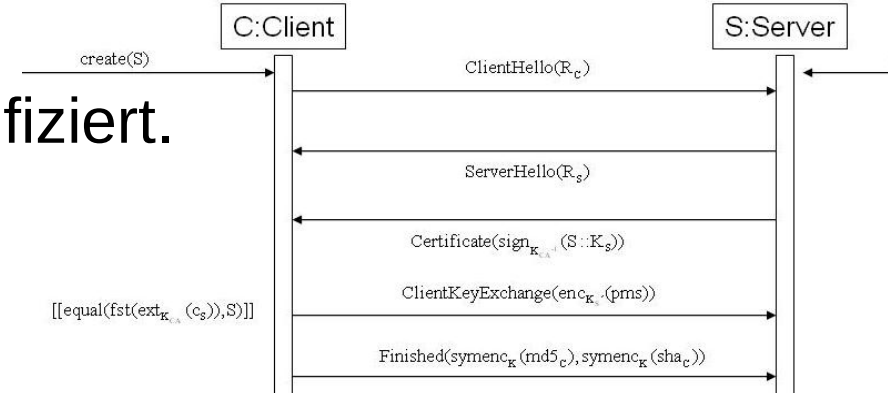
Anwendung: Java Secure Sockets Extension



Verschiedene Versionen der Java-Bibliothek
"Java Secure Sockets Extension (JSSE)"; open-source Re-Implementierung (Jessie).

Mehrere Schwachstellen identifiziert.

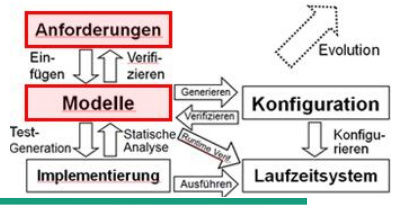
Auch **größere Evolution** (Re-Implementierung) !



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	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. $S \rightarrow C$: Certificate[X509Cert$_s$]	2	2	1.474
S4. C : Veri(X509Cert$_s$)	2	2	3.854
...	...	...	...
Total of 7 messages and 3 checks	27	86	40.303

Beispielanforderung: Sicherer Informationsfluss



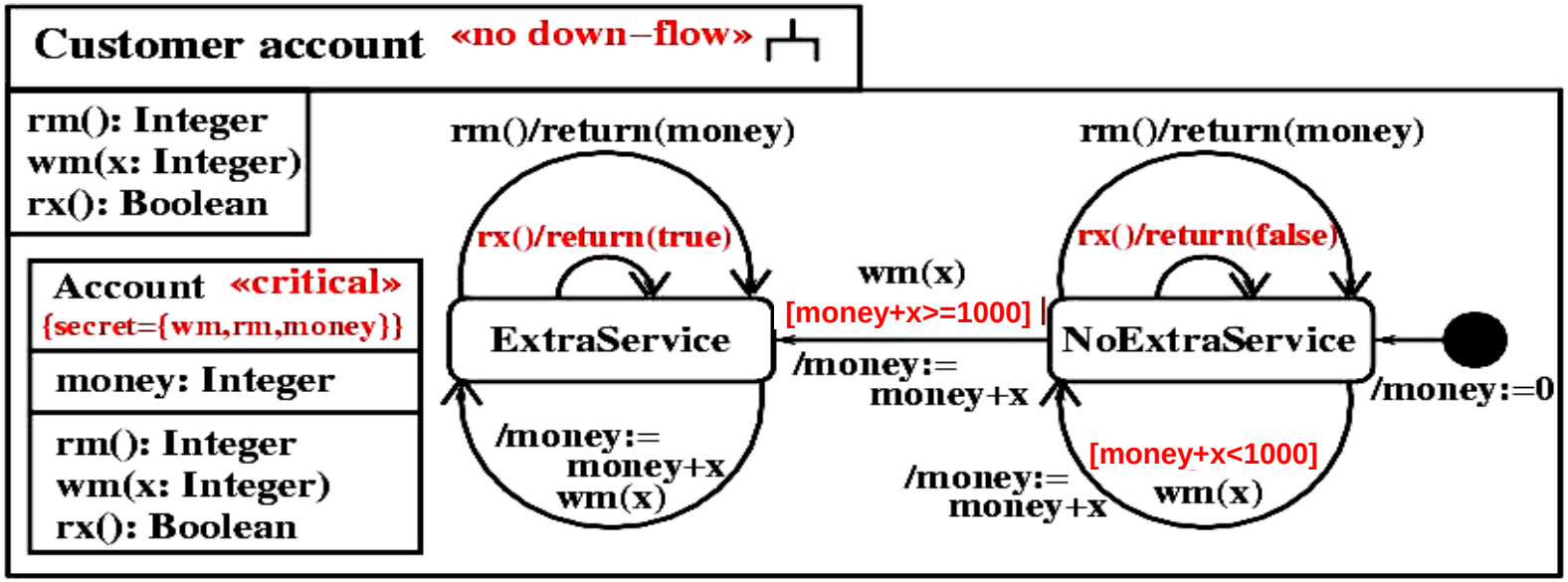
Beispiel „sicherer Informationsfluss“: Kein Informationsfluss von „vertraulich“ zu „nicht-vertraulich“ (insbes. indirekt oder partiell !).

Analyse:

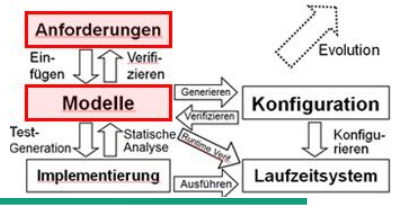
Systemzustände unterscheiden sich nur in vertraulichen Variablen

➔ Auswirkungen auf nicht-vertrauliche Variablen nicht unterscheidbar

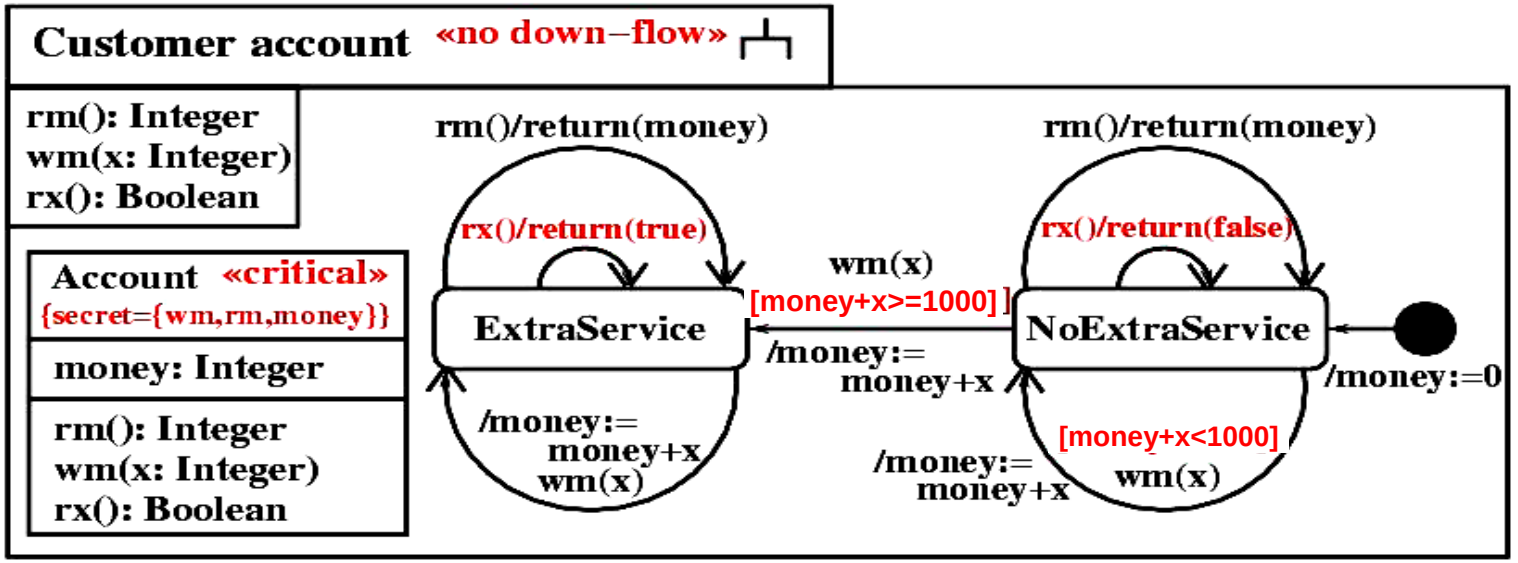
Beispiel: Web-basiertes Kundenkonto. Sicher ?



Beispiel: Sicherer Informationsfluss

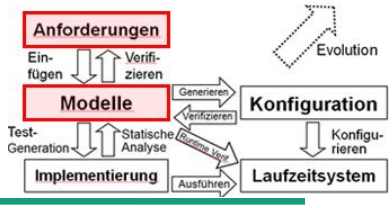


Unsicher: vertrauliches Attribut *money* beeinflusst Rückgabewert der nicht-vertraulichen Methode *rx()*.



Wie kann man diese Anforderung **automatisch** und in Gegenwart von **Evolution verifizieren** ?

Sichere Evolution auf Modellebene: Analyse durch Formalisierung



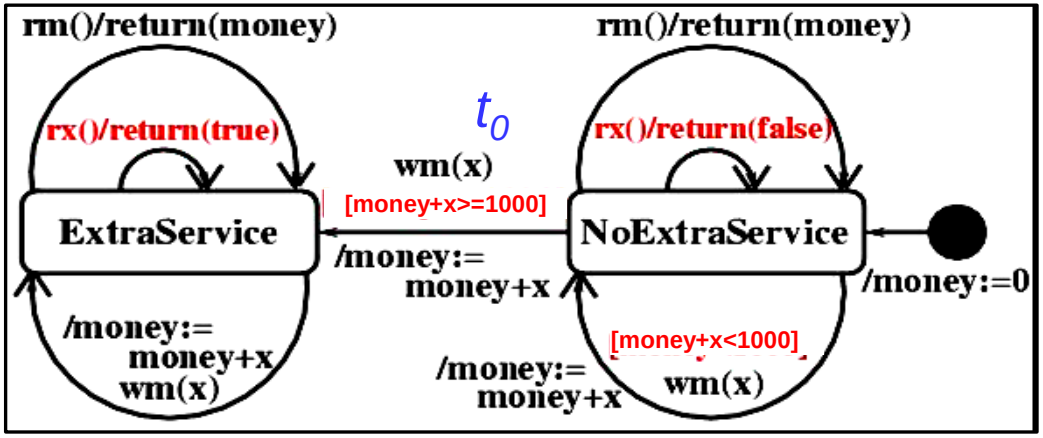
Formalisierung der Modellausführung. Für Statechart-Transition *t=(source,msg,cond[msg],action[msg],target)* und erhaltene Nachricht *m*, Formalisierung der Ausführung der Transition:

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

(wobei: $state_{current}$ aktueller Zustand;
$state_{current.t(m)}$ Zustand nach Ausführung von *t* mit Nachricht *m*).

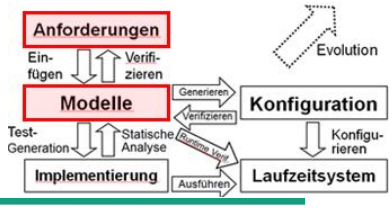
Beispiel: Transition t_0:

[Ochoa, Jürjens, Cuellar. Non-interference on UML State-charts. TOOLS Europe '12]



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

Beispiel sicherer Informationsfluss: Automatische Verifikation



Systemzustände unterscheiden sich nur in vertraulichen Variablen

➔ Auswirkungen auf nicht-vertrauliche Variablen nicht unterscheidbar:

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

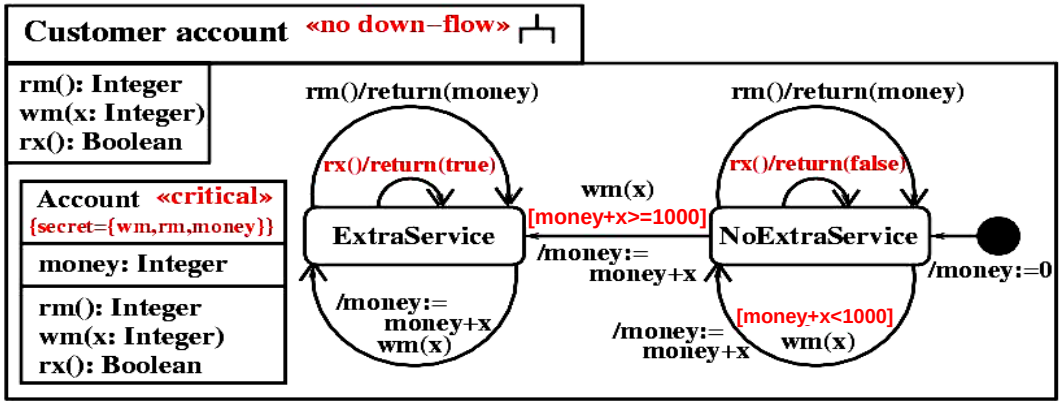
($state_{current} \approx_{pub} state'_{current}$: unterschiedlich nur in vertraulichen Variablen;
$state_{current.t(m)}$: nächster Zustand nach Transition t)

Beispiel: Unsicher: vertrauliches Attribut *money* beeinflusst Rückgabewert der nicht-vertraulichen Methode *rx()*.

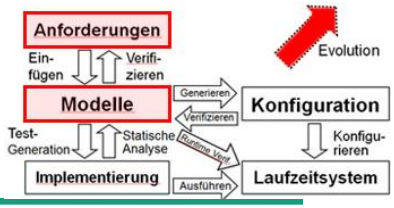
ExtraService $\approx_{pub}$ *NoExtraService*

aber nicht:

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



Sicherheit vs. Verfeinerung: Problem

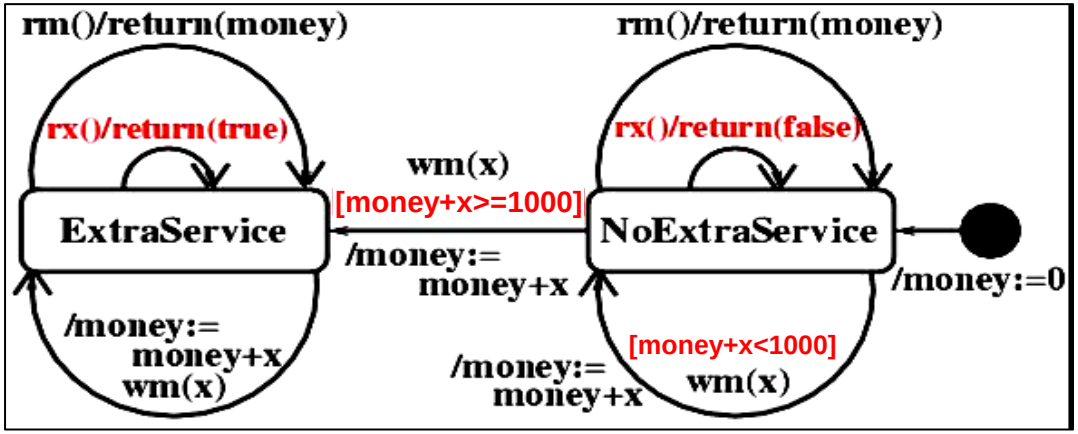


Verhaltensbewahrende Verfeinerung: würde Bewahrung von Sicherheitsanforderungen erwarten.

„**Verfeinerungsparadox**“: Im Allgemeinen jedoch nicht ! [Roscoe‘96].

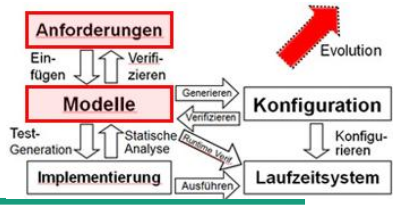
Obiges Beispiel: Transitionen *rx()/return(true)* (bzw. *false*)

Verfeinerung der „sicheren“ Transition *rx()/return(random_bool)*.



Problematisch: Vermischung von Nichtdeterminismus zur Unterspezifikation bzw. als Sicherheitsmechanismus.

Sicherheit vs. Verfeinerung: Lösung



Unser Spezifikationsansatz **trennt** beide Nichtdeterminismus-Arten.

Resultat: Verfeinerung bewahrt wichtige Sicherheitseigenschaften.

Beispiel:

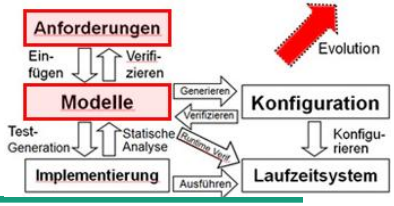
Definition Q refines P ($P \rightsquigarrow Q$) if for each $\vec{s} \in \mathbf{Stream}_{I_F}$ have $[\![P]\!](\vec{s}) \supseteq [\![Q]\!](\vec{s})$.

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

Nachweis: mit formaler Semantik.

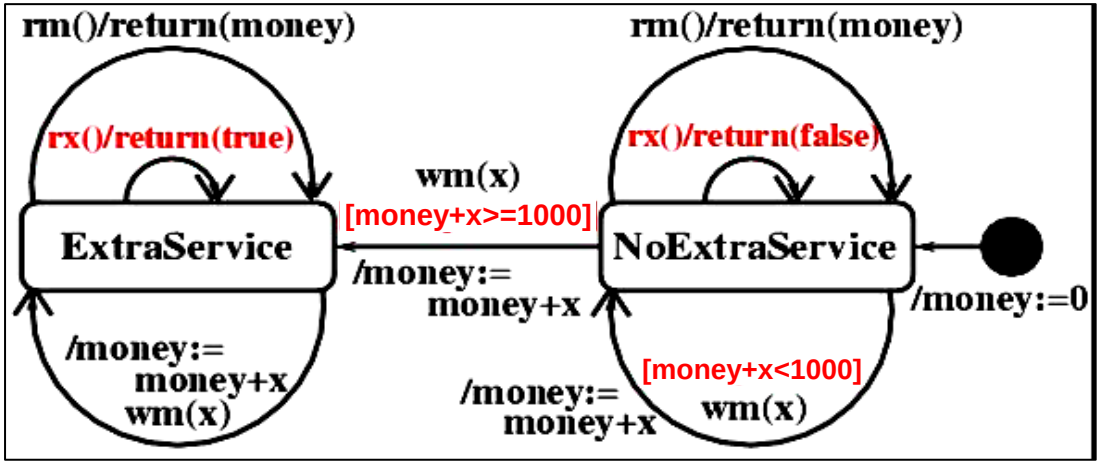
Obiges Beispiel: mit unserem Ansatz **keine** Verfeinerung.

Sicherheit vs. Modularisierung: Problem



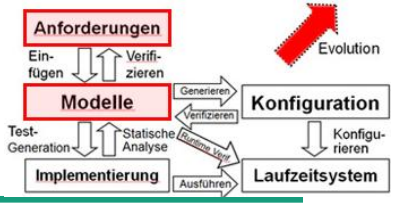
Problem: Sicherheitseigenschaften i.A. **nicht kompositional**.

Obiges Beispiel: Zustände *ExtraService* und *NoExtraService* jeweils „sicher“ (nur ein Rückgabewert von *rx*); Komposition in Statechart nicht.



Frage: Unter welcher Bedingung bewahrt Komposition Sicherheit ?

Sicherheit vs. Modularisierung: Lösung



Lösungsansatz: Sicherheitsanforderung als „Rely-guarantee“[1]-Bedingung definieren.

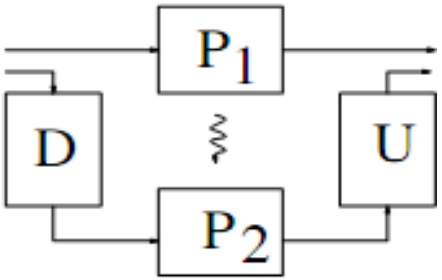
Resultat: Dafür Bedingungen an Kompositionalität.

Beispiel: Modulare Dekomposition mit **Schnittstellen** D, U

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.

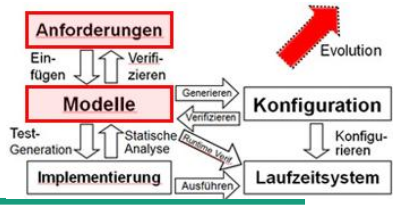
Nachweis: mit formaler Semantik.

Obiges Beispiel: Rely-guarantee-Formalisierung zeigt: Sichere Komposition nicht möglich.



[1] C.B. Jones 1981

Evolutions-basierte Verifikation: Beispiel Sicherer Informationsfluss

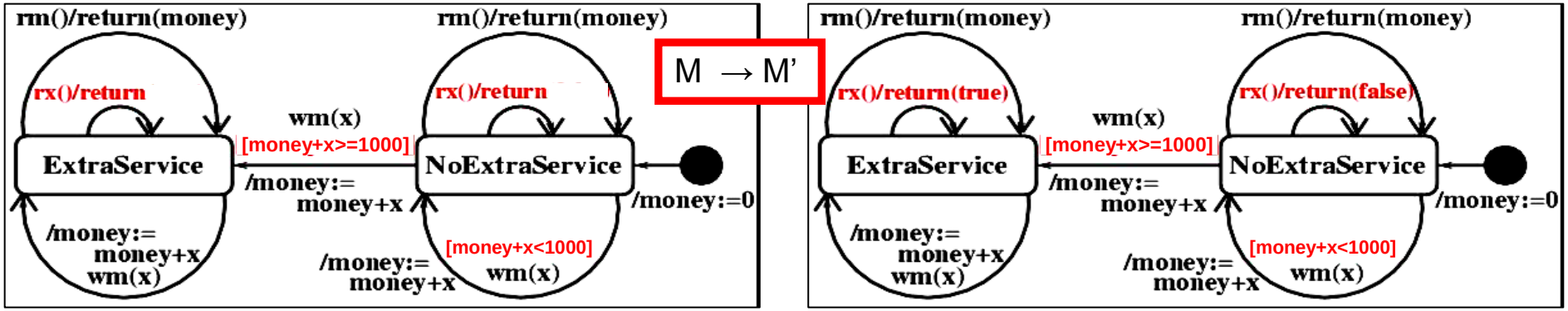


Sicherer Informationsfluss:

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

Evolution M → M‘: Nur Systemzustände betrachten, für die:

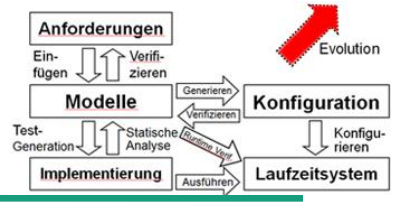
- $state_{current} \approx_{pub} state'_{current}$ in M‘, aber nicht in M, oder
- $state_{current.t(m)} \approx_{pub} state'_{current.t(m)}$ in M, aber nicht in M‘.



Beispiel: $wm(0).rx() \approx_{pub} wm(1000).rx()$ in M aber nicht in M‘.

➔ M‘ **verletzt** sicheren Informationsfluss.

Validierung

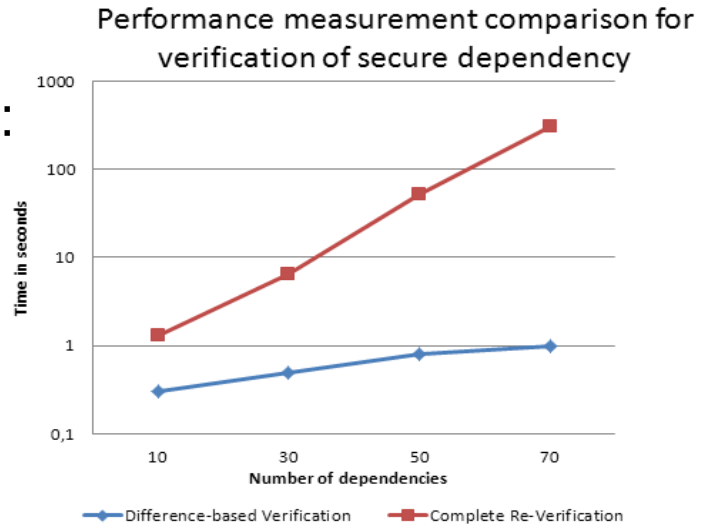


- **Korrektheit**: mittels formaler Semantik
- **Vollständigkeit**: jede Modell-Transformation darstellbar als Folge von Löschen, Ändern und Hinzufügen von Modell-Elementen

Performanzgewinn am **größten**, wenn **Differenz << Software.** Beispiel-Resultat:

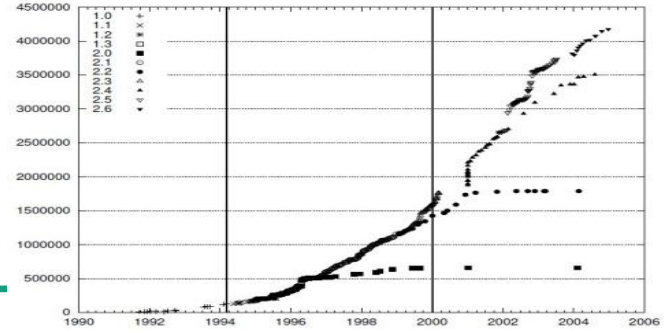
- Komplette Re-Verifikation: Laufzeit **exponentiell** in Softwaregröße
- Evolutions-basierte Verifikation: Laufzeit **linear** in Softwaregröße

(jeweils **gleichbleibende Differenzgröße**).

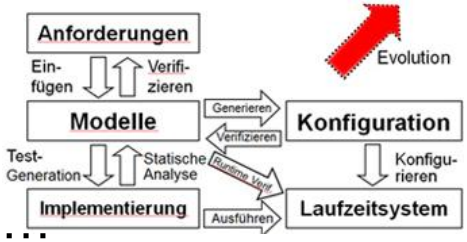


Ist erfüllt:

- **Wartung stabiler Software-Versionen**
- **änderungsbegleitende QS** (nightly builds)



Zusammenfassung:
Sichere Evolution für verteilte Software-Systeme

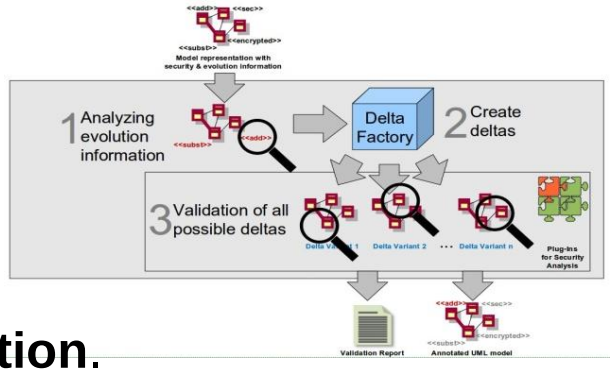


Evolution: Herausforderung für Qualitätssicherung.

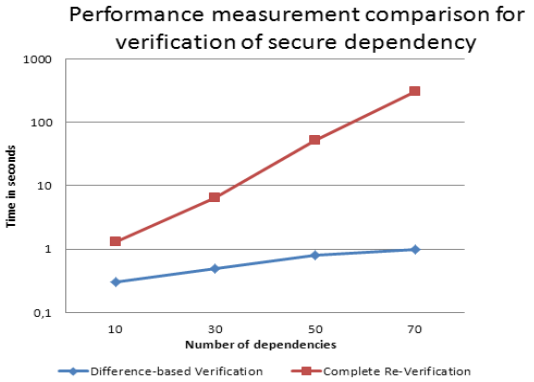
Frage: Wann QS-Ergebnisse wiederverwenden ?

Resultate: Bedingungen für Bewahrung von Anforderungen…

- … durch Design-/Architektur-Prinzipien (z.B. **Verfeinerung**, **Modularisierung**).
- Damit **Evolutions-basierte Verifikation**:
  - **Modellanalyse**
  - **statische Code-Analyse** / **Run-time Verification**.



- Implementiert in **Werkzeugen**: erhebliche Performanzverbesserung.



Validierung: Erfolgreicher Einsatz in der Praxis.

Aktuell: Softwaremigration in Clouds, Übertragung auf Software-Produktlinien.