Grundstrukturen fehlertoleranter und sicherer verteilter Systeme

Felix Gärtner

TU Darmstadt

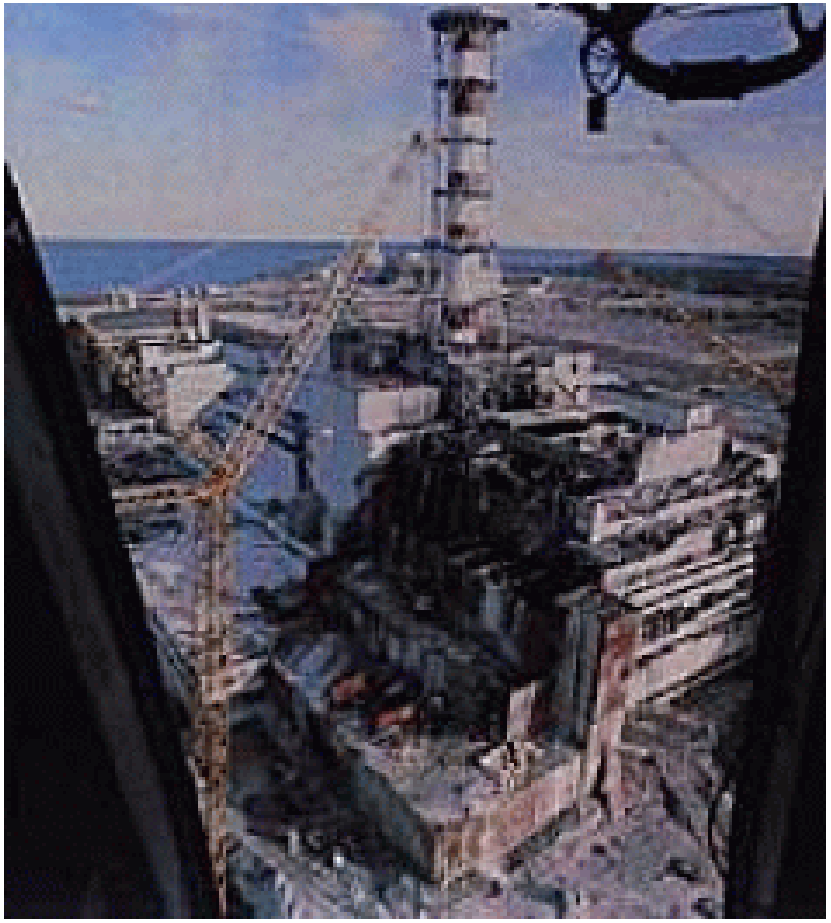
felix@informatik.tu-darmstadt.de

Korrekte Systeme?

Quelle: http://www.mpia-hd.mpg.de/SUW/SuW/1996/10-96/S713Abb1.html

Es funktioniert nicht, obwohl alles so ist wie erwartet!

Fehlertolerante Systeme?



Quelle: http://www.1a-homepage-werbung.de/wahnwitz/Atomkraft-Hysterie/chern2.jpg

Es funktioniert nicht, trotz redundanter Steuerungen!

Sichere Systeme?

Quelle: http://www.rp-online.de/news/multimedia/netzreporter/2001-0115/

Es funktioniert nicht, trotz Virenscanner!

Kritische Infrastrukturen

Quelle: http://www.cs.virginia.edu/~survive/

- Computersysteme schaffen Flexibilität aber auch neue Gefahren und Abhängigkeiten.
- U.S.-Strategen beschwören bereits “elektronisches Pearl Harbor.”

Ziel: Theoretisch abgesicherte Ingenieursmethodik

- Computersysteme müssen verläßlich sein (*dependable* [Laprie 1992]).
- Verläßliche Systeme zu bauen ist schwer! [Neumann 1995]
- These: Nur eine theoretisch abgesicherte Ingenieursmethodik kann "die Welt retten".

Quelle: http://www.af.mil/photos/images/00179c.jpg

Hier eigentlich: Karikatur von H. Maurer: "One day the internet broke down"

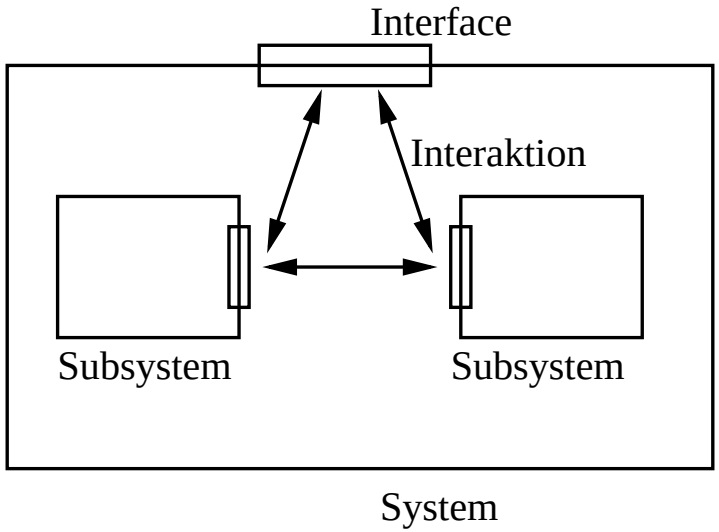
Grundlage einer Methodik

- Grundlage einer Methodik:
  - Rigorosität durch Formalisierung: Modellierung.
  - Verständnis praktischer Systeme durch Untersuchung der Modelle.
- Herausarbeiten von grundlegenden Abhängigkeiten (*tradeoffs*), Möglichkeiten und Unmöglichkeiten (Schwierigkeiten)
- "Angebote an die Ingenieursintuition": Bewußtsein für inhärente Schwierigkeiten und Fallstricke

Systeme und ihre Eigenschaften

- Wir wollen, daß Systeme gewisse Eigenschaften erfüllen:
- Eigenschaften verläßlicher Systeme (*dependable systems* [Laprie 1992]):
  - Verfügbarkeit (*availability*)
  - Zuverlässigkeit (*reliability*)
  - Sicherheit (*safety* und *security*)
- Wie modellieren wir diese Eigenschaften adäquat? Wie weisen wir sie nach?

Zwischenruf: Zwei Systemsemantiken



- Modellierung von System und Eigenschaften:
  - *interleaving*-Semantik: Ereignisse im System sind atomar und total geordnet.
    - Temporale Logik, Model Checking, Statecharts
  - *partial order*-Semantik: Ereignisse des Systems sind atomar und "halbgeordnet".
    - Petri Netze, Message Sequence Charts, Raum/Zeitdiagramme

Gliederung

- Motivation
- Thema: Arbeiten zum Thema Fehlertoleranz
  - *interleaving*-Semantik: Entwurfstheorie fehlertoleranter Systeme und die Rolle von Redundanz.
  - *partial order*-Semantik: Beobachtbarkeit unter Störeinflüssen (*crash* Fehlern).
- Thema: Ansätze im Bereich *security*
  - Unterschiede zur Fehlertoleranz.
  - Anstätze und Problemstellungen.
- Zusammenfassung und Ausblick

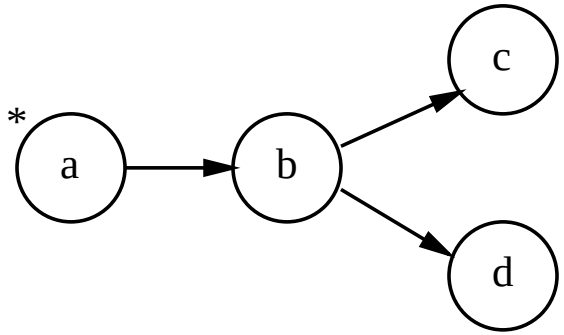
Fehlertoleranz

Über eine konstruktive Entwurfstheorie für fehlertolerante Systeme
und die Rolle von Redundanz

Systemmodell

- Verteiltes System aus Rechnerknoten, die mittels Nachrichtenaustausch kommunizieren.
- Nachrichten können beliebig lange unterwegs sein.
- Rechnerknoten können beliebig langsam sein.
- Gebräuchliches Systemmodell für das Internet.
- Effekte verschiedener Systemparameter (Fehler, Randomisierung, Uniformität) recht gut verstanden.

Reaktive Verteilte Systeme



- Programm Σ: Automat (C, I, T) mit Zustandsmenge C, Menge initialer Zustände $I \subseteq C$ und Zustandsübergangsrelation $T \subseteq C \times C$.
- Automat erzeugt Abläufe (*traces*), z.B. abc, abd.
- Semantik $sem(\Sigma)$ des Programms Σ: Menge aller Abläufe von Σ (*interleaving* Semantik).

Eigenschaften reaktiver Systeme

- Eigenschaft P: Menge von Abläufen.
- Beispiele:
  - Eigenschaft "niemals d" modelliert als die Menge

$$\{a, aa, aaa, b, ba, baa, cab, \ldots\}$$

  - Eigenschaft "wann immer a, dann im nächsten Schritt b (falls es einen nächsten Schritt gibt)":

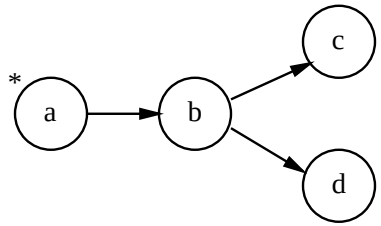
$$\{ab, a, bb, ccc, cccab, dbababb, \ldots\}$$

- Temporale Logik als "Syntax" für Eigenschaften:
  - Beispiel: $\Diamond a = \{a, ba, aaa, bbbbba, dcdca, \ldots\}$
  - anderes Beispiel: $\Box\neg d =$ "immer nicht d" $=$ "niemals d"

Nachweis der Eigenschaft

- Gegeben ein Programm Σ und eine Eigenschaft P.
- Σ erfüllt P falls alle Abläufe von Σ in P enthalten sind.
- Beispiel: Erfüllt das Programm die folgende Eigenschaft?

$$P = \{abcd, aabcd, abc, abbc, abd, abad, \ldots\}$$



- Formal: Implikation (Teilmengenbeziehung) als Korrektheitsbegriff ($sem(\Sigma) \subseteq P$)
- Nachweis je nach Art der Eigenschaft

Sicherheitseigenschaften (*safety*)

- Sicherheitseigenschaft S ("immer . . . "): Beispiele
  - wechselseitiger Ausschluß: "niemals zwei Prozesse gleichzeitig im kritischen Abschnitt"
  - partielle Korrektheit: "wenn das System terminiert ist, dann gilt im System die Nachbedingung"

- Formal: Verletzung passiert durch etwas unwiderruflich "Schlechtes"

$$\sigma \notin S \Rightarrow \exists i. \forall \beta. \sigma|_i \cdot \beta \notin S$$

- Nachweis mittels Invariante.

Lebendigkeitseigenschaften (*liveness*)

- Lebendigkeitseigenschaft L ("schlußendlich . . . "): Beispiele
  - Terminierung: "schlußendlich wird der Terminierungszustand erreicht"
  - Verfügbarkeit: "jeder *request* wird schlußendlich beantwortet"

- Formal: Das "Gute" ist immer möglich

$$\forall i.\exists \beta.\sigma|_i \cdot \beta \in L$$

- Nachweis mit Terminierungsfunktion.

- Eigenschaftsklassen sind fundamental [Alpern and Schneider 1985]:

$$\forall P.\exists S, L.P = S \cap L$$

Fehlertoleranzspezifikationen [Arora and Kulkarni 1998; Gärtner 1999]

- Fehler werden als Programmtransformation (Σ nach Σ') modelliert.
- Beispiele:
  - Wertefehler: füge "falsche" Zuweisung ins Programm ein.
  - Zeitfehler: baue Endlosschleife ein.
  - Analogie: Softwarebasierte Fehlerinjektion.
- Generell: Fehler können zu einer Verletzung von S oder L führen.
- Semantische Klassifikation der Arten von Fehlertoleranz:
  - Maskierend: Σ' erfüllt $S \cap L$
  - Nicht-Maskierend: Σ' erfüllt $\Diamond S \cap L$
  - *fail-safe*: Σ' erfüllt S

Fehlertoleranzkomponenten

- Fehlertolerantes Programm =
  fehler-intolerantes Programm + Fehlertoleranzkomponenten
- Detektor: erkennt Systemzustand
- Korrektor: erzwingt Systemzustand
- Abstraktionen von vielen bekannten Fehlertoleranzmechanismen, gute Strukturierungsmöglichkeit [Gärtner 1998].

Fehlertoleranztheorie

- Theoreme [Arora and Kulkarni 1998]:
  - Detektoren sind notwendig und hinreichend für Sicherheitseigenschaften.
    - Hinreichend: Man kann ein System fehlertolerant bzgl. einer Sicherheitseigenschaft machen durch Hinzufügung von Detektoren.
    - Notwendig: Wenn ein System durch Hinzufügung von Komponenten fehlertolerant bzgl. einer Sicherheitseigenschaft gemacht wurde, dann enthält es Detektoren.
  - Korrektoren sind notwendig und hinreichend für Lebendigkeitseigenschaften
    - ...

- Beispiele:
  - *checkpointing/recovery*: Detektor erkennt Fehlzustand, Korrektor erzwingt konsistenten Zustand.
  - *triple modular redundancy*: Detektor erkennt Inkonsistenz der Replikate, Korrektor berechnet "korrekten" Wert und zwingt diesen Wert den Replikaten auf (im *voter*).

Redundanz [Gärtner and Völzer 2001]

- Funktionsprinzip: Redundanz
  - Platzredundanz = nicht-erreichbarer Zustand
    - Fehlererkennungszustände (erreichbar nur, wenn Fehler auftreten)
    - Verallgemeinerung von fehlererkennenden Codes
  - Zeitredundanz = nicht-ausführbare Transition
    - Fehlerbehebungsaktionen (Transitionen, aus einem Fehlerzustand herausführen)
    - Abstraktion von Fehlerbehebungsmechanismen

- Resultate:

fehlertolerant bzgl.	notwendig
Sicherheit	Platzredundanz
Lebendigkeit	Platzredundanz und Zeitredundanz

- Verständnis für die grundlegende Wirkungsweise von fehlertoleranten Systemen in *interleaving* Semantik.

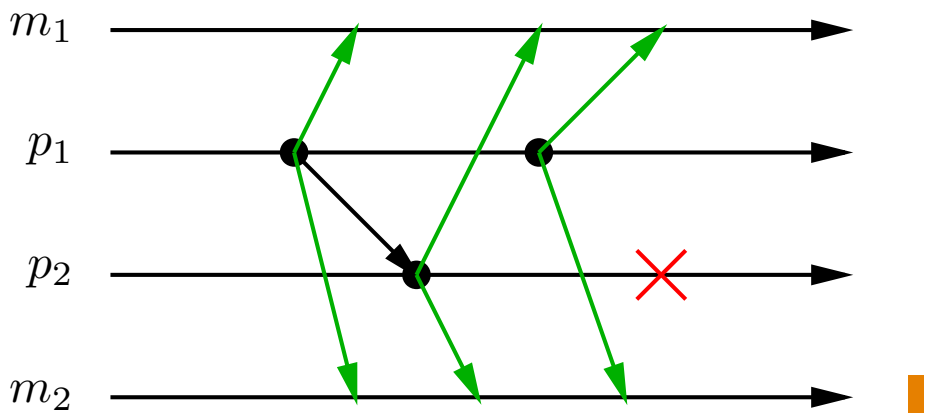
Fehlertoleranz

Über die Schwierigkeiten, verteilte Systeme mit Fehlern zu beobachten

Achtung: jetzt Halbordnungssemantik!

Beobachten fehlerbehafteter verteilter Systeme

- Beobachten = globale Prädikate entdecken unter der Annahme von *crash* Fehlern.



- Gilt ein globales Prädikat φ in der Berechnung?

- Halbordnungssemantik hat zur Folge, daß Frage nach Gültigkeit eines allgemeinen Prädikates nicht sinnvoll ist (Beobachterabhängigkeit) [Schwarz and Mattern 1994].

- Einschränkung auf einen Anwendungsprozeß und einen Beobachter macht die Sache zunächst einfacher.

Entdeckungssemantiken

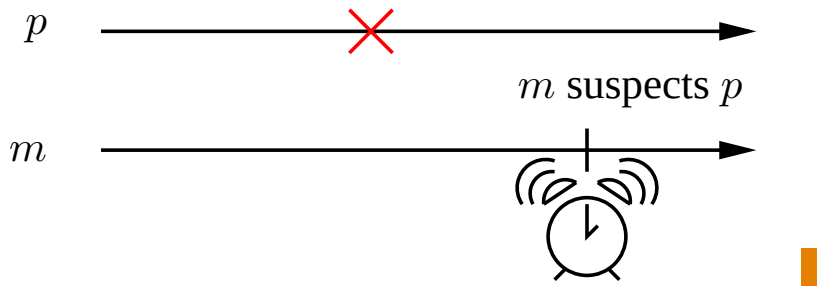
- Drei Semantiken:
  - Perfekte Prädikatsentdeckung Sem_1:
    - (S) Wenn der Algorithmus die Gültigkeit meldet, dann galt φ vorher mal in der Berechnung.
    - (L) Wenn φ gilt, dann wird der Algorithmus schlußendlich die Gültigkeit von φ melden.
  - Stabilisierende Prädikatsentdeckung Sem_2: L und $\Diamond S$.
  - *best effort* Prädikatsentdeckung Sem_3: L und $\Box\Diamond S$.

- Beispiel:
  - $\varphi \equiv$ "p ist abgestürzt als er eine Sperre (*lock*) hielt"
  - Sem_1 wünschenswert (keine Fehlalarme) aber oft nicht realisierbar.
  - Endliche Anzahl von Fehlalarmen mit Sem_2.
  - Falls φ niemals gilt, garantiert Sem_3 immerhin, daß die Gültigkeit von φ nicht "dauerhaft" gemeldet wird (Sem_3 ist also besser als nichts).

- Wie kommt man an die "Absturzinformationen"?

Fehlerdetektoren

- Abstrakter Mechanismus, um Aussagen über den operationellen Zustand eines Nachbarprozesses zu machen.



- Eigenschaften:
  - Prozess p wird niemals verdächtigt bevor er abstürzt.
  - Falls p abstürzt, wird p schlußendlich auch einmal verdächtigt.

- Sowas nennt man einen perfekten Fehlerdetektor.

- NB: Nur an der Schnittstelle definiert, keine Aussage über Implementierung.

Unzuverlässige Varianten

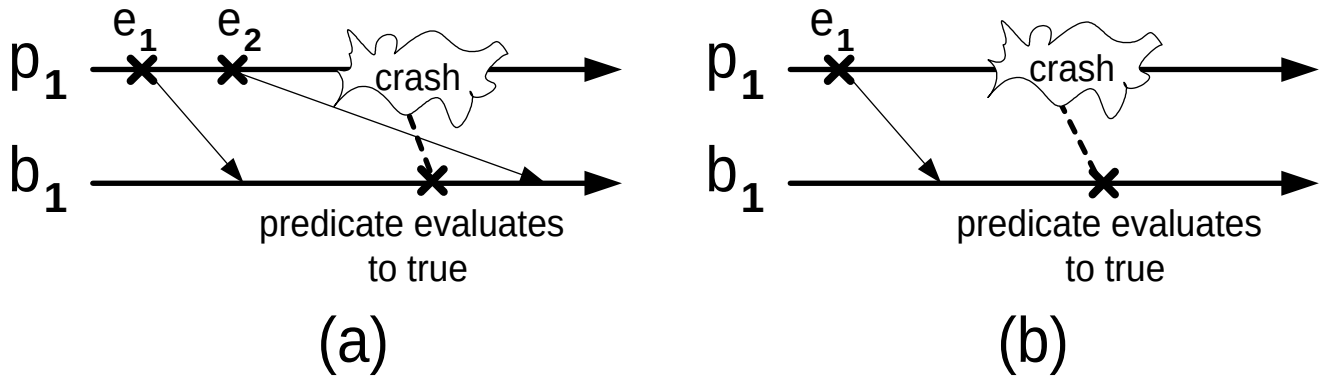
- In der Praxis gibt es keine perfekten Fehlerdetektoren, nur imperfekte.
- Modellieren durch abgeschwächte Eigenschaften an der Schnittstelle:
- Nach endlicher Zeit perfekt (*eventually perfect*) [Chandra and Toueg 1996]:
  - Es gibt eine Zeit nachdem kein Prozeß verdächtigt wird bevor er abstürzt.
- *best effort* Fehlerdetektoren [Garg and Mitchell 1998]:
  - Prozesse, die nie abstürzen, werden nicht dauerhaft verdächtigt.
- Frage: Wie kann man (unzuverlässige) Fehlerdetektoren einsetzen, um Prädikatserkennung zu implementieren?

Resultate

- Man kann stabilisierende Prädikatsentdeckung Sem_2 implementieren mit einem nach endlicher Zeit perfekten Fehlerdetektor.

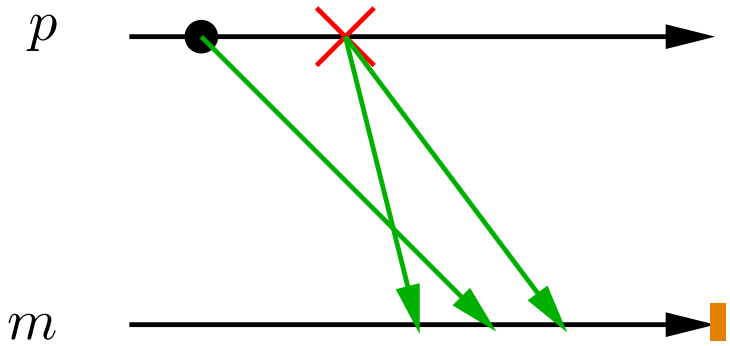
- Man kann *best effort* Prädikatserkennung Sem_3 implementieren mit einem *best effort* Fehlerdetektor.

- Aber: Ein perfekter Fehlerdetektor ist nicht hinreichend, um perfekte Prädikatsentdeckung Sem_1 zu implementieren.



Stärkerer Fehlerdetektor

- Benötigte Eigenschaft:
  - Man muß den Absturz von p in Beziehung setzen zu den Nachrichten, die möglicherweise noch von p unterwegs sind.
  - Informationen über den Nachrichtenkanal fehlen.



- Man kann einen solchen Fehlerdetektor aber implementieren!

Verallgemeinerung für n Prozesse

- Bisher nur das Szenario mit einem Anwendungsprozess und einem Beobachter behandelt, jetzt n Prozesse und m Beobachter.
  - Fragen der Beobachterabhängigkeit klären!
    - Prädikate einschränken (z.B. nur stabile Prädikate betrachten).
    - Fragestellung ändern (nicht "Gilt das Prädikat?" sondern "Könnte es jemand beobachten?") [Gärtner and Kloppenburg 2000].
  - Beispiel: perfekte Fehlerdetektoren können perfekte Prädikatsentdeckung Sem_1 erreichen, falls φ stabil ist.

- Lehren:
  - Prädikatsentdeckung ist schwierig selbst mit perfekten Fehlerdetektoren.
  - In der Praxis muß man oft mit stabilisierender Prädikatsentdeckung auskommen.

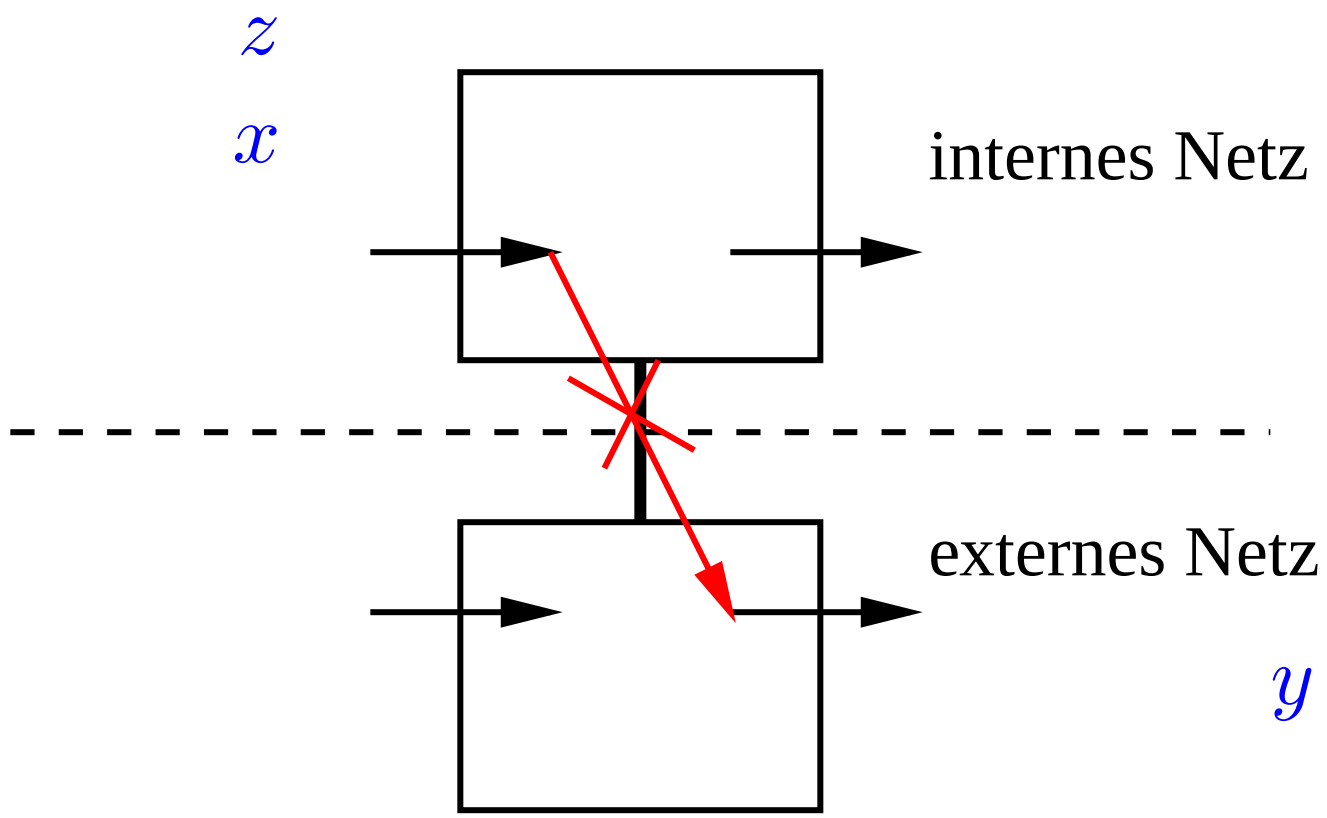
Security

Über die Schwierigkeiten, *security* zu definieren. . .

Was ist *security*?

- Auch Sicherheit- und Lebendigkeit:
  - Sicherheitseigenschaft: Zugriffskontrolle [Schneider 1998], Vertraulichkeit [Gray, III. and McLean 1995], Integrität.
  - Lebendigkeitseigenschaft: Verfügbarkeit.
- Fehlertoleranzmethodik einsetzbar (da sie nur auf abstrakten Eigenschaften definiert worden ist).
  - *security* Eigenschaften als Menge von Abläufen definieren.
  - Angriff ausprogrammieren.
  - Nachweis der Eigenschaft auf üblichem Weg.
- Probleme mit Informationsfluß über verdeckte Kanäle (relevant in Hochsicherheitsbereichen), z.B. träge Firewall.

Informationsflußeigenschaften



Mögliche Abläufe: x, y z, y

Eigenschaften von Eigenschaften

- Eigenschaften der Art: falls Ablauf x, y möglich ist, dann muß auch Ablauf z, y möglich sein.
- Abschlußeigenschaft einer Menge:

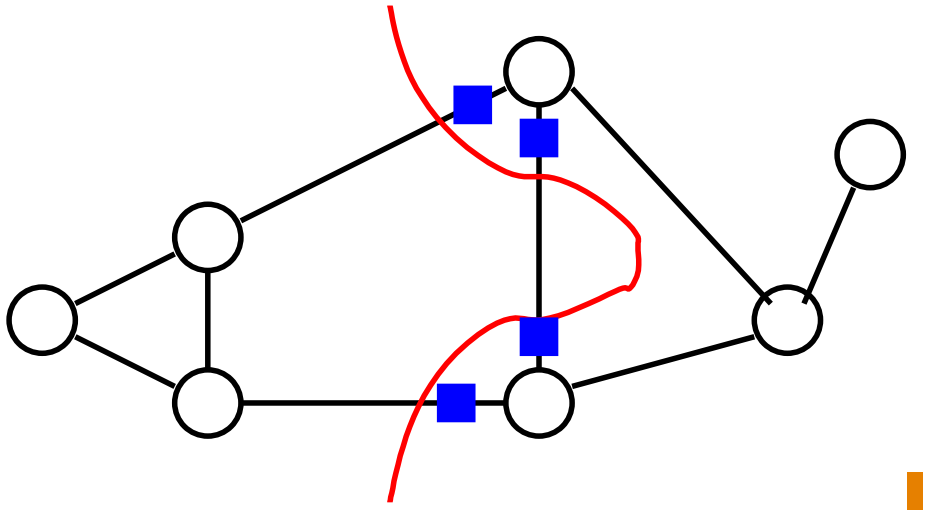
$$\sigma \in P \Rightarrow f(\sigma) \subseteq P$$

- Keine Eigenschaft, sondern Eigenschaft einer Eigenschaft, Menge einer Menge von Abläufen.
- Fachbegriff: *noninterference*.
- *security* = Sicherheit, Lebendigkeit und *noninterference*

security-Komponenten

- Detektoren, Korrektoren für Sicherheits- und Lebendigkeitseigenschaften (Zugriffskontrolle, Verfügbarkeit, etc.).

- Aufteilung des Zustandsraums in Vertraulichkeitszonen, Kontrolle des Informationsflusses auf den Grenzen.



- Protektor für *noninterference* (Abstraktion von Verschlüsselung und Firewall).

security-Theorie: Ansätze

- Formalisierungen:
  - Wissenstheoretische (diskrete) Formalisierung von Vertraulichkeitszonen [Halpern and Moses 1990].
- Mögliches Theorem:
  - Protektoren hinreichend und notwendig für *noninterference*.
- Viele offene Fragen!

Fragen und Projekte

- Offene Fragen für die Forschung:
  - Definition eines Protektors? Funktionsprinzipien? Kompositionsprinzipien?
  - Beziehungen zu kryptographischen (komplexitätstheoretischen) Definitionen von *security*?
  - Aber Fehler $\neq$ Angriffe:
    * Fehler treten zufällig auf, Angriffe nicht.
    * Neue Angriffe verlassen oft das Systemmodell (*timing attacks*, *versioning attacks*, *social attacks*), Komplexität des Modells steigt.
    * Wahrscheinlichkeit als Maß der Fehlerabdeckung ungenügend (eher "Aufwand" eines Angreifers).

- Ansätze:
  - Untersuchung verdeckter Kanäle: "gehackter" Webserver mit verdecktem Kanal.
  - Abwehrmaßnahmen: *lock keeper* [Haffner et al. 2000]
  - Ausführungsplatform mit Sicherheitszellen: UMLinux, VMWare

Einsichten

- Sonderstellung von Vertraulichkeit (unerwünschtem Informationsfluß) als Eigenschaft!
- Vertraulichkeit läßt sich (im allgemeinen) nicht verfeinern [McLean 1994]!
- Vertraulichkeit teilweise im Widerspruch zu Sicherheits und Lebendigkeitseigenschaften (Flexibilität vs. Restriktionen).
- Oft: Fehlertoleranzmethodik anwendbar, ähnliche Entwurfsmuster verwenden!

Ausblick

- *Security* macht Computersysteme noch komplexer als sie sowieso schon sind.
- Keine einzelne Person weiß, wie und warum auch nur kleine Systeme funktionieren (Reparatur durch Reset, Handauflegen, etc.).
- Akademische Informatik muß ihren Beitrag zu einer theoretisch abgesicherten aber trotzdem benutzbaren Methodik leisten.
  - Konflikt zwischen theoretischer Fundierung und Brauchbarkeit in der Praxis nicht unterschätzen!
- Ausblick (These): Gesamtkomplexität wird auf absehbare Zeit nicht beherrschbar werden (*ubiquitous computing*):
  - Müssen mit Risiken leben lernen: Einfache *fallback*-Lösungen.
  - Müssen mit Komplexität leben lernen: Verhaltensstudium von Computersystemen, "Soziologie" der Informatik

Danksagungen

- Folien produziert unter Verwendung von pdfLaTeX und Klaus Guntermanns PPower4.

References

Alpern, B. and Schneider, F. B. 1985. Defining liveness. *Information Processing Letters 21*, 181–185.

Arora, A. and Kulkarni, S. S. 1998. Component based design of multitolerant systems. *IEEE Transactions on Software Engineering 24*, 1 (Jan.), 63–78.

Chandra, T. D. and Toueg, S. 1996. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM 43*, 2 (March), 225–267.

Garg, V. K. and Mitchell, J. R. 1998. Distributed predicate detection in a faulty environment. In *Proceedings of the 18th IEEE International Conference on Distributed Computing Systems (ICDCS98)* (1998).

Gärtner, F. C. 1998. Fundamentals of fault tolerant distributed computing in asynchronous environments. Technical Report TUD-BS-1998-02 (July), Darmstadt University of Technology, Darmstadt, Germany. To appear in *ACM Computing Surveys*, 31(1), March 1999.

Gärtner, F. C. 1999. An exercise in systematically deriving fault-tolerance specifications. In *Proceedings of the Third European Research Seminar on Advances in Distributed Systems (ERSADS)* (Madeira Island, Portugal, April 1999).

Gärtner, F. C. and Kloppenburg, S. 2000. Consistent detection of global predicates under a weak fault assumption. In *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS2000)* (Nürnberg, Germany, Oct. 2000), pp. 94–103. IEEE Computer Society Press.

Gärtner, F. C. and Völzer, H. 2001. Defining redundancy in fault-tolerant computing. In *Brief Announcement at the 15th International Symposium on DIStributed Computing (DISC 2001)* (Lisbon, Portugal, Oct. 2001).

Gray, III., J. W. and McLean, J. 1995. Using temporal logic to specify and verify cryptographic protocols. In *Proceedings of the Eighth Computer Security Foundations Workshop (CSFW '95)* (June 1995), pp. 108–117. IEEE Computer Society Press.

Haffner, E.-G., Engel, T., and Meinel, C. 2000. Integration der Schleusentechnologie "Lock-Keeper" in moderne Sicherheitsarchitekturen. In M. Schumacher and R. Steinmetz Eds., *Sicherheit in Netzen und Medienströmen. Tagungsband des GI-Workshops "Sicherheit in Mediendaten"*, Informatik Aktuell (2000), pp. 17–26. Springer-Verlag.

Halpern, J. Y. and Moses, Y. 1990. Knowledge and common knowledge in a distributed environment. *Journal of the ACM 37*, 3 (July), 549–587.

Laprie, J.-C. Ed. 1992. *Dependability: Basic concepts and Terminology*, Volume 5 of *Dependable Computing and Fault-Tolerant Systems*. Springer-Verlag.

McLean, J. 1994. A general theory of composition for trace sets closed under selective interleaving functions. In *Proceedings of the 1994 IEEE Symposium on Research in Security and Privacy* (Oakland, CA, 1994), pp. 79–93.

Neumann, P. G. 1995. *Computer Related Risks*. ACM Press.

Schneider, F. B. 1998. Enforceable security policies. Technical Report TR98-1664 (Jan.), Cornell University, Department of Computer Science, Ithaca, New York.

Schwarz, R. and Mattern, F. 1994. Detecting causal relationships in distributed computations: in search of the holy grail. *Distributed Computing 7*, 149–174.

Zusammenfassung

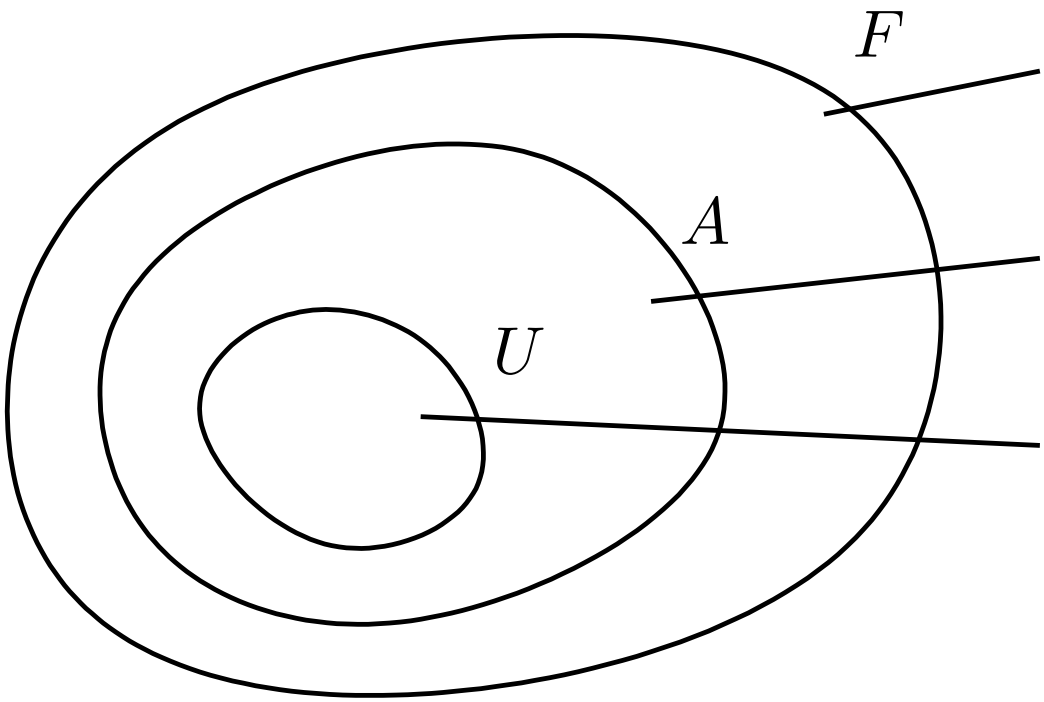
Verteilte Systeme (wie zum Beispiel Rechnernetze) müssen sicher sein, d.h. einerseits robust gegenüber dem zufälligen Versagen einzelner Bauteile und andererseits geschützt gegen absichtliche Störeinwirkungen. Um Sicherheit zu erreichen, muß man darum sowohl Methoden der Fehlertoleranz anwenden, als auch Methoden aus dem Bereich der “computer security”. So ähnlich die Ziele beider Bereiche scheinen, so unterschiedlich sind die Methoden, die jeweils angewendet werden. Dieser Vortrag beschreibt ein Projekt, in dessen Rahmen die Grundlage einer einheitlichen Entwurfsmethodologie für fehlertolerante und sichere (secure) verteilte Systeme untersucht werden soll. Im ersten Teil des Vortrags wird zunächst auf bereits weitgehend abgeschlossene Arbeiten im Bereich der Grundlagen fehlertoleranter Systeme eingegangen. Vorgestellt wird eine einheitliche Entwurfsmethodik für fehlertolerante Systeme sowie eine Analyse der grundsätzlichen Wirkungsprinzipien von Fehlertoleranzverfahren. Der zweite Teil des Vortrags beschreibt eine Menge von Ansätzen, die Ziele und Methoden der Fehlertoleranz und der “computer security miteinander zu vergleichen.

Zusatzfolien

Notwendigkeit von Platzredundanz

- $\Sigma = (C, I, T, A)$, Fehlermodell F, Eigenschaft P.
- Σ erfüllt P, $F(\Sigma)$ verletzt P.
- Baue Σ' aus Σ so daß $F(\Sigma')$ erfüllt P.
- Dann hat Σ' nicht erreichbare Zustände.
- Beweis: Σ' besitzt eine Transition, die auch in Σ zur Verletzung von P geführt hätte.

Beweis: Informationsfluß ist keine Menge von Abläufen



assumed trace set
absence of info flow

set of all unsuccessful traces is fair

restriction of A to all traces that give away the item