Formale Grundlagen der Fehlertoleranz in verteilten Systemen

Felix Gärtner

17. Mai 2001

Abhängigkeit von Computersystemen

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

Sicherheit und Versicherungen

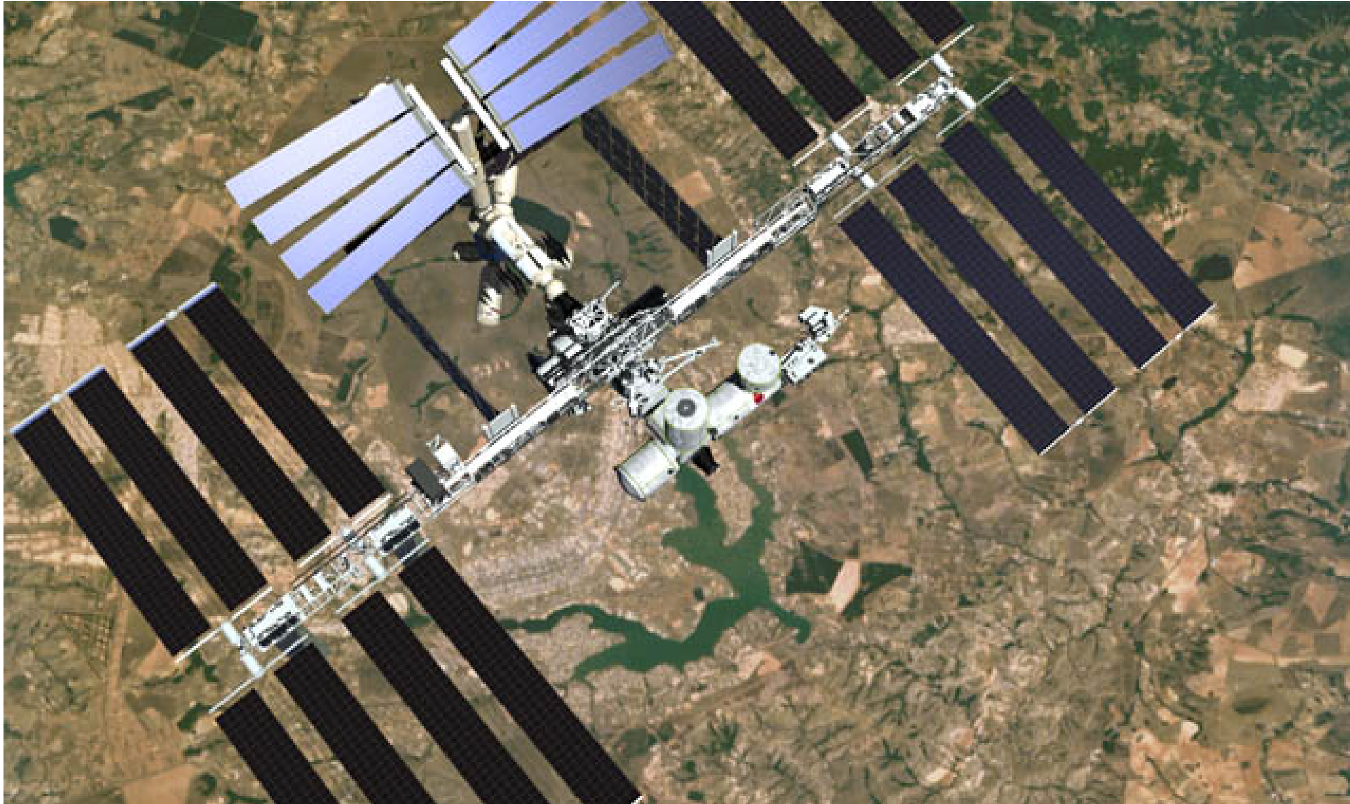
- Wir alle machen Fehler — auch Computer!

  In the future, the computer security industry will be run by the insurance industry.

  Bruce Schneier

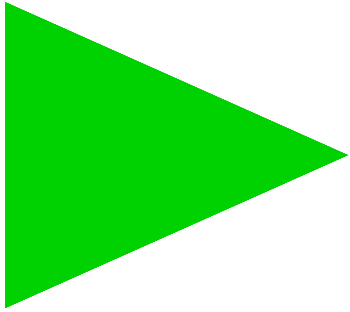
- Fehler tolerieren durch Risikomanagement (z.B. Risikovermeidung) und Technologie (z.B. qualitative Software, Fehlertoleranz).

Grenzen des Risikomanagements



Quelle: http://spaceflight.nasa.gov/

- Soft-/Hardwarefehler unvermeidbar — wer versichert die ISS?



Forschungsagenda

- Analyse durch mathematische Modellierung.
  - Beitrag: Systemmodelle der Fehlertoleranz.
- Verständnis/Intuition entwickeln für Wirkungsweise von existierenden Methoden der Fehlertoleranz.
  - Beitrag: Begrifflichkeit der Redundanz.
- Grundlagen legen für die Ingenieurspraxis.
  - Beitrag: Methoden der Beobachtungen.

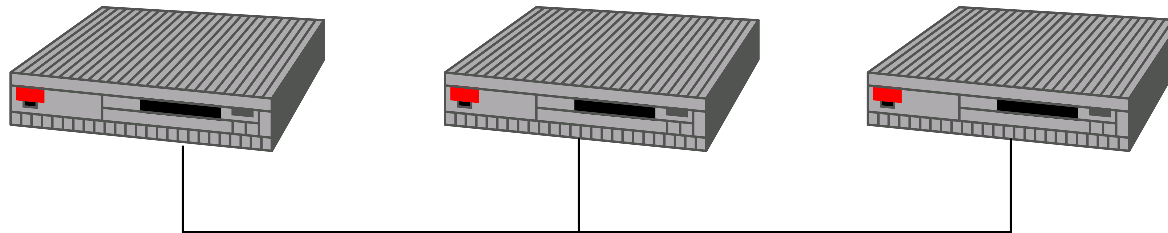
Das asynchrone Systemmodell

- Geographisch verteilte Rechner kommunizieren über Nachrichten.
- Nachrichten können lange unterwegs sein.
- Computer können sehr langsam werden.
- Gebräuchliches Modell für das Internet.
- Computer können abstürzen.

Sicherheit und Lebendigkeit

- Beschreibung funktionaler Eigenschaften.
- Zwei grundlegende Klassen von Eigenschaften verteilter Systeme:
  - Sicherheit (*safety*), z.B. partielle Korrektheit, wechselseitiger Ausschluss, “niemals x”.
  - Lebendigkeit (*liveness*), z.B. Terminierung, “schlußendlich x”.

Beispiel: Übereinstimmungsproblem



- Annahme: maximal ein Computer rechnet falsch.

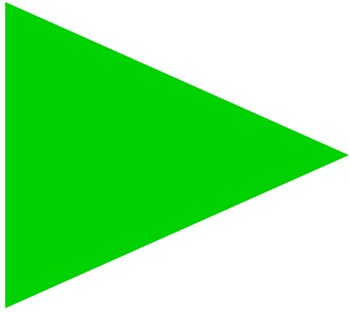
- Ziel: Computer sollen jede neue Berechnung mit dem gleichen Startwert beginnen.

- Problem ist nicht immer (einfach) lösbar!

Beliebte Fehlertoleranzmethoden

- *Triple modular redundancy* (TMR).
- *Checkpointing/recovery*.
- *Automatic repeat request* (ARQ).

Was sind die grundlegenden Mechanismen, die in Fehlertoleranzverfahren eine Rolle spielen?



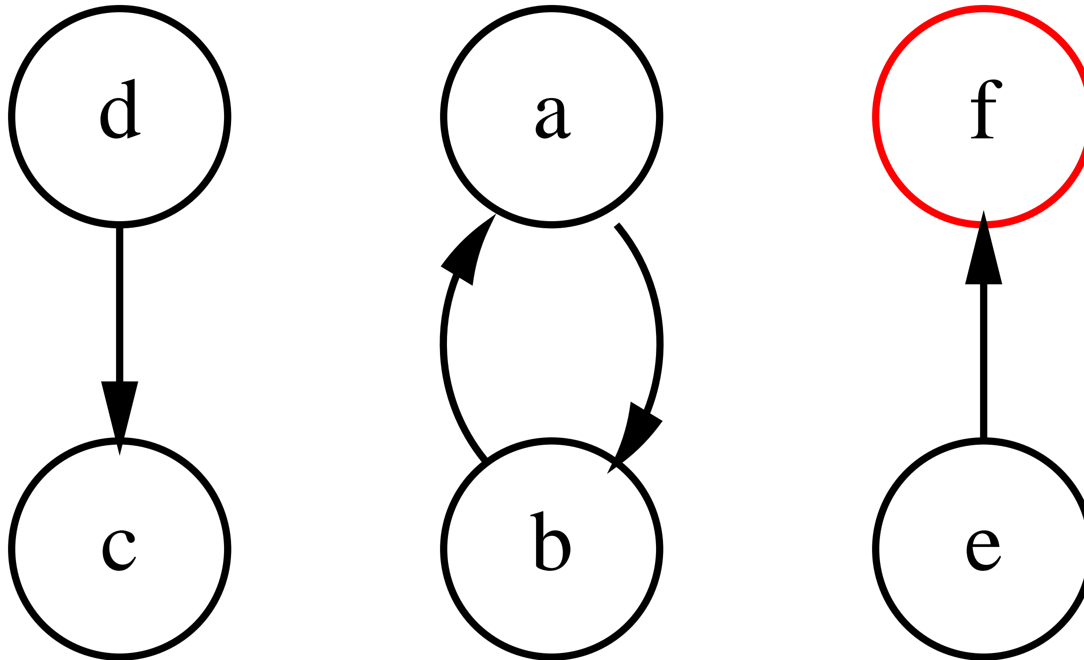
Stand der Forschung

- Fehlertoleranztheorie von Arora und Kulkarni.
- Fehlertolerantes Programm =
  Programm + Fehlertoleranzkomponenten.
- Detektor: erkennt einen bestimmten Systemzustand.
- Korrektor: stellt einen bestimmten Systemzustand her.
- Abstraktion vieler gebräuchlicher
  Fehlertoleranzmethoden.

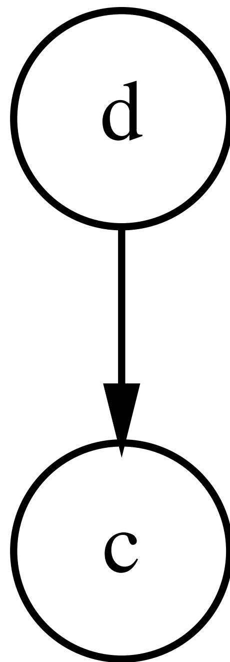
Grundlagen der Fehlertoleranz

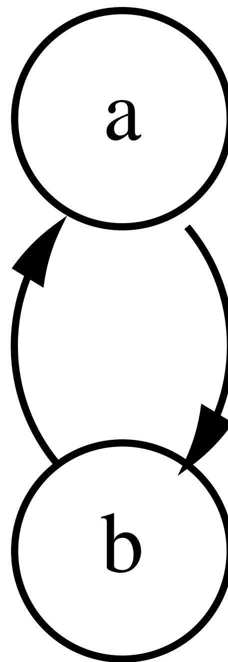
- Theoreme:
  - Detektoren sind notwendig und hinreichend, um Sicherheit zu erreichen.
  - Korrektoren sind notwendig und hinreichend, um Lebendigkeit zu erreichen.
- Korrektor enthält immer einen Detektor.
- Detektoren und Korrektoren sind abstrakte Gebilde. Wie funktionieren sie?

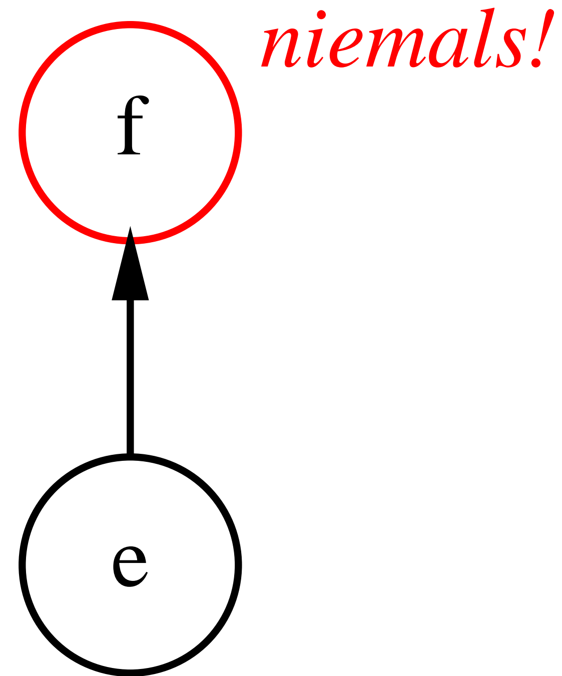
Modellierung als Automaten



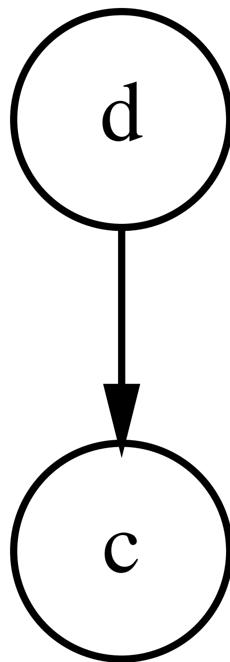
Modellierung als Automaten

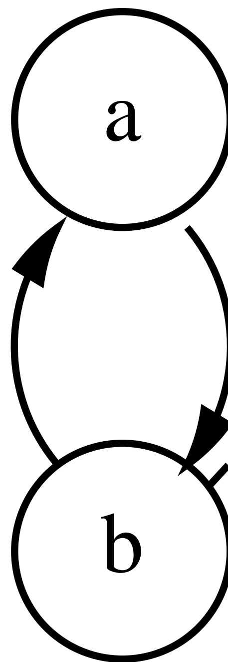

d
c

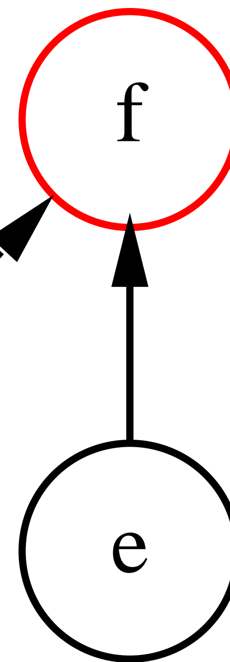

a
b


niemals!
f
e

Modellierung als Automaten

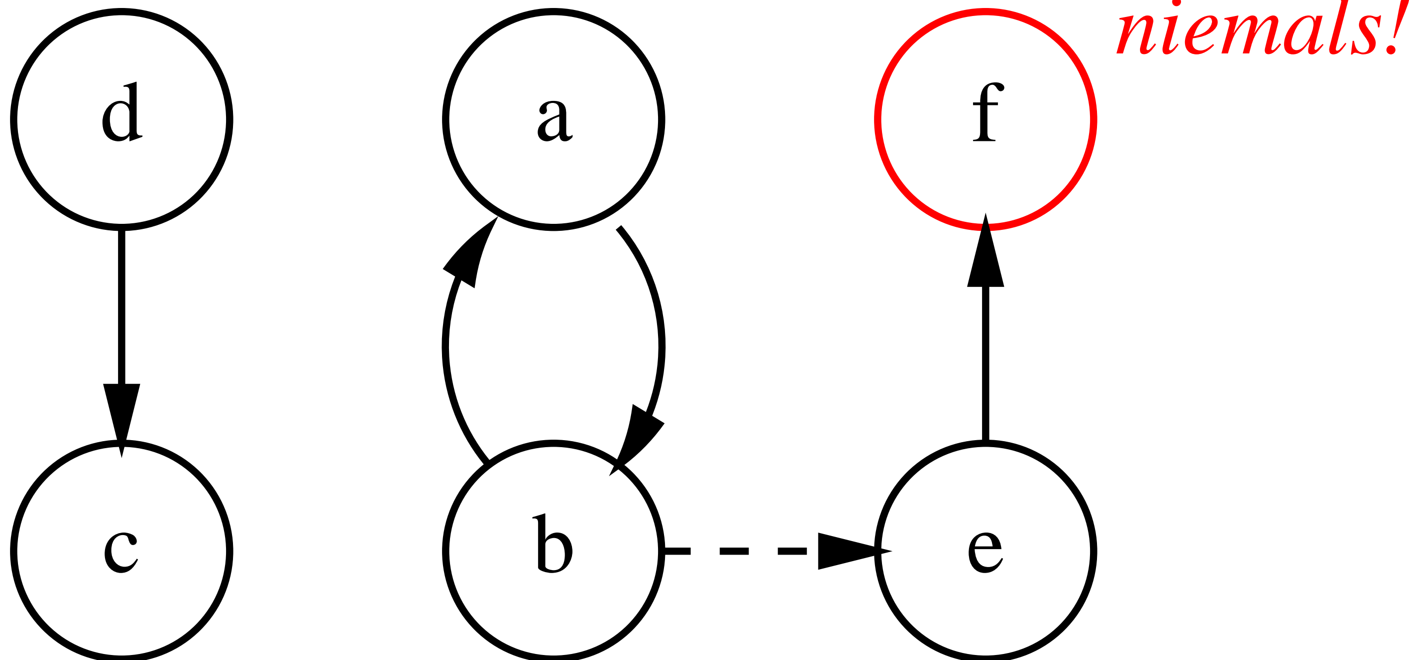




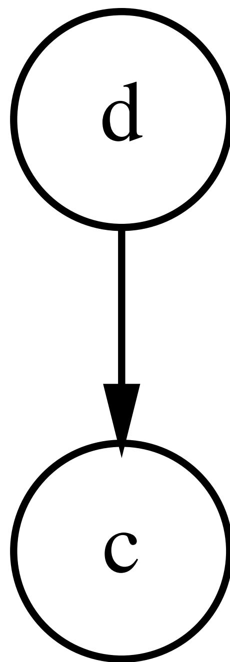


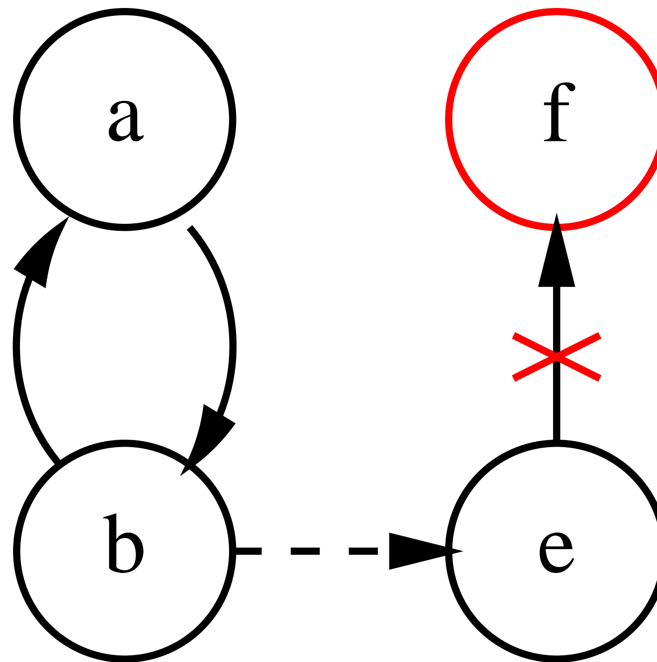
niemals!

Modellierung als Automaten

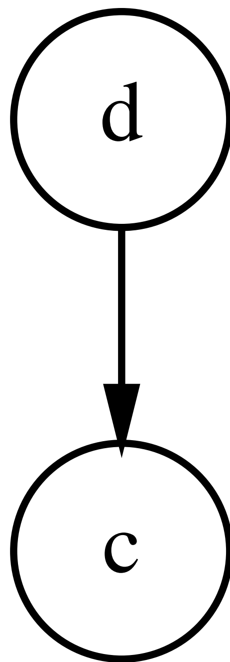


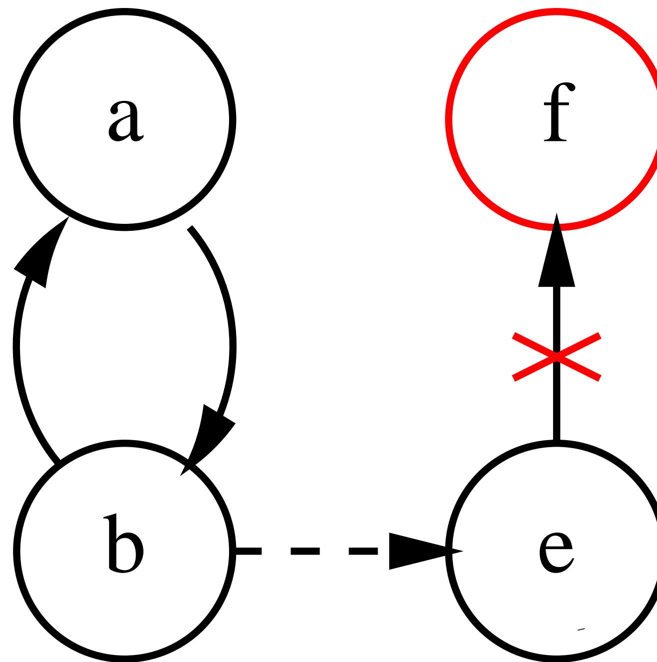
Modellierung als Automaten

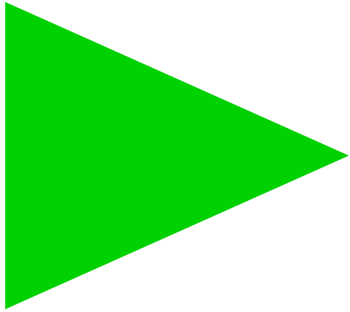




Modellierung als Automaten



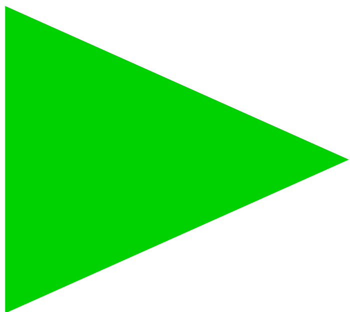




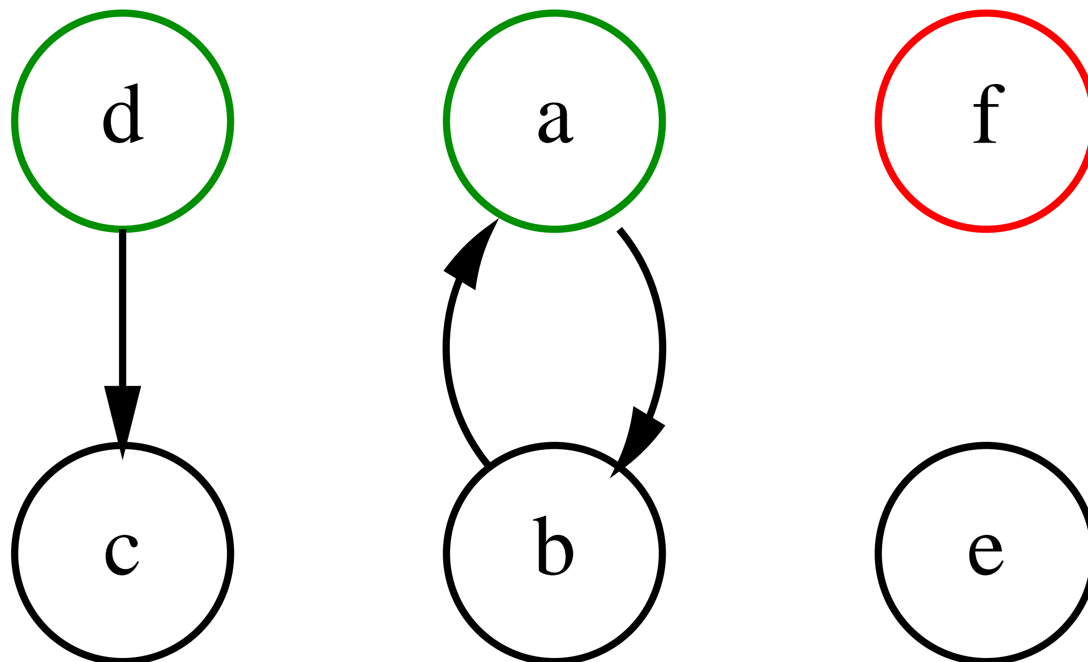
Speicherredundanz

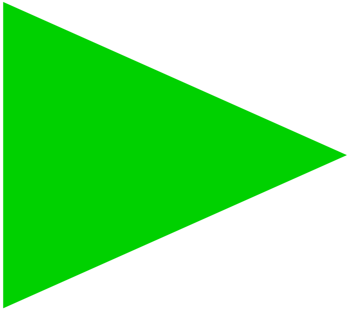
redundante Zustände = Speicherredundanz
Zustände, die nicht erreicht werden,
wenn keine Fehler auftreten.

- Bekannt aus der Kodierungstheorie.
- Notwendige “Puffer” zur Fehlererkennung.
- Einsicht: Detektoren enthalten Speicherredundanz.

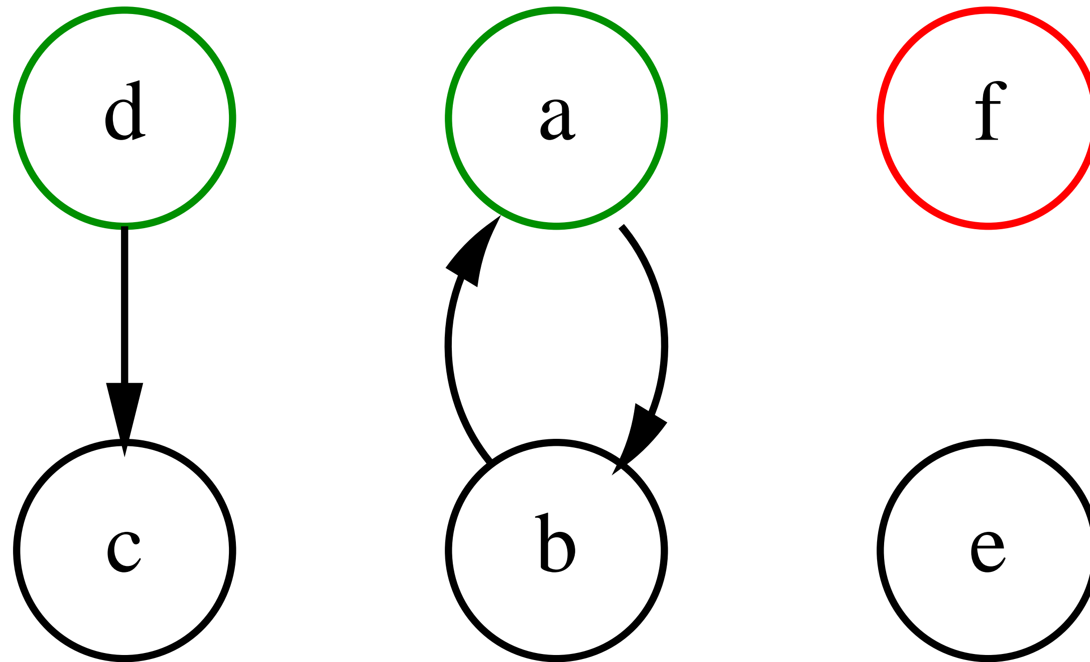


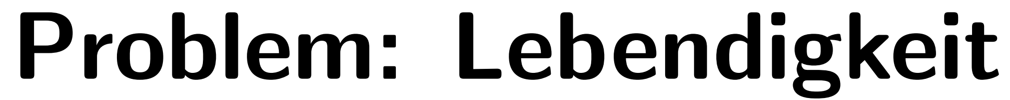
Problem: Lebendigkeit



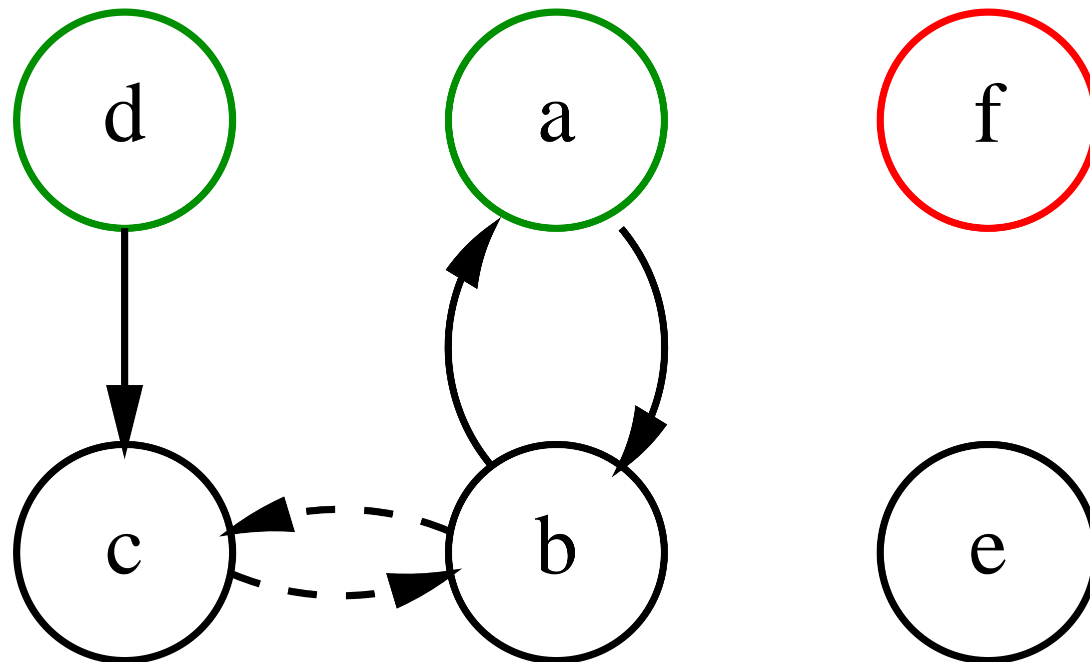


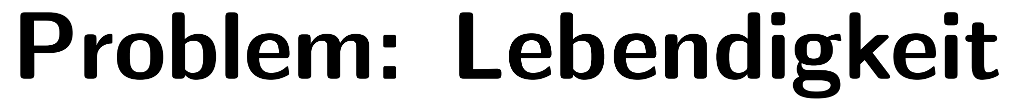
Problem: Lebendigkeit

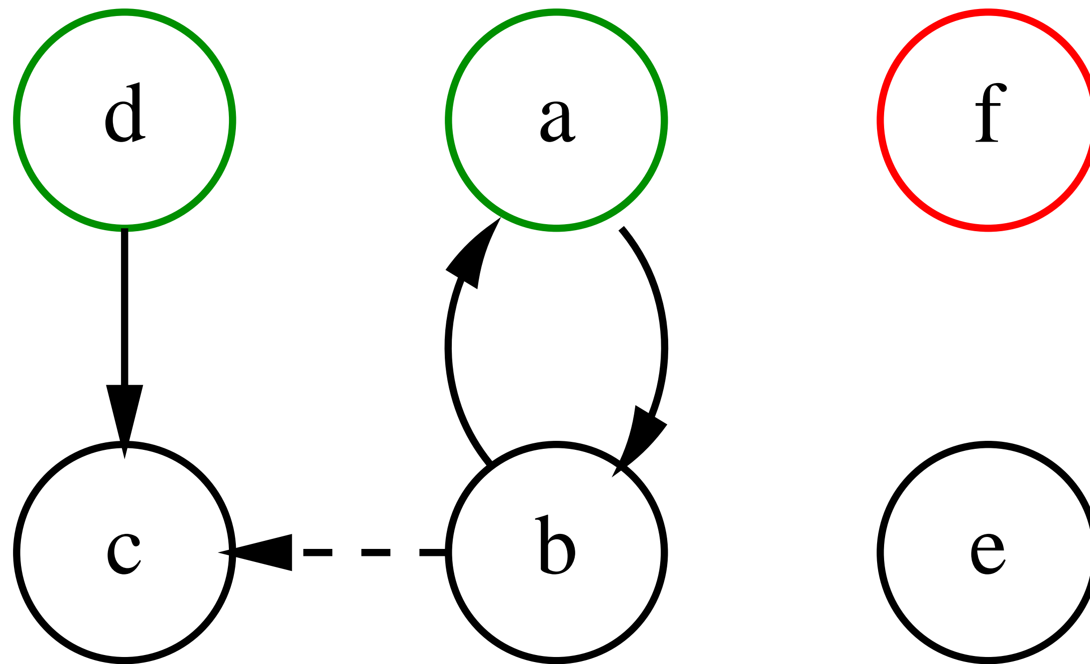




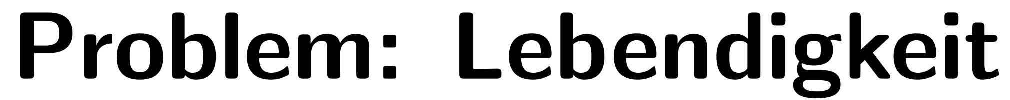
immer wieder!

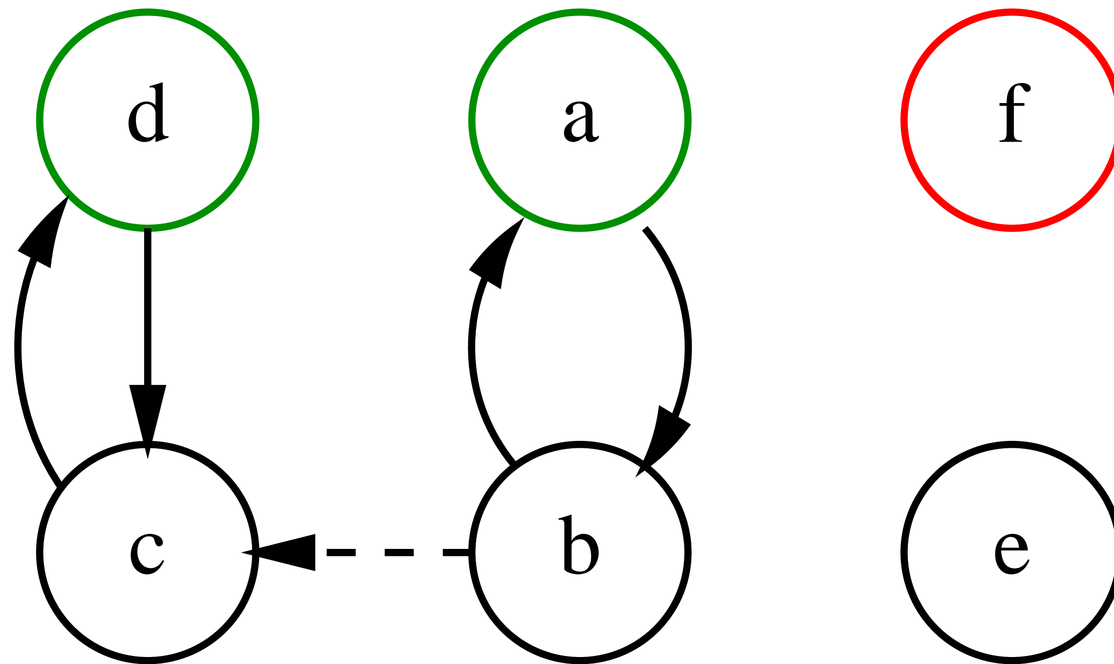


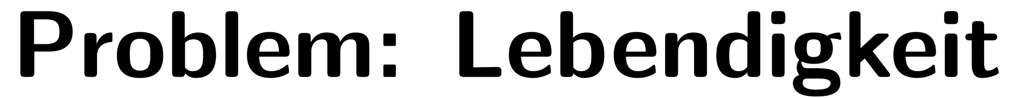

Problem: Lebendigkeit


immer wieder!
d
a
f
c
b
e

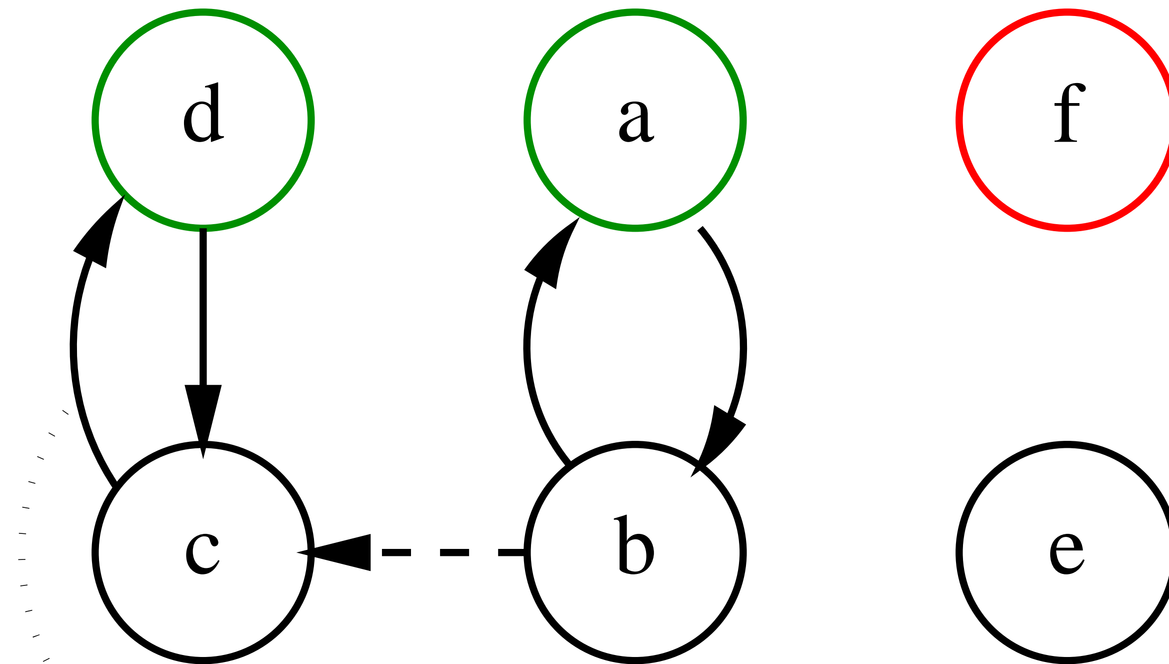
Problem: Lebendigkeit







immer wieder!



Zeitredundanz

Zeitredundanz

redundante Transitionen = Zeitredundanz
Transitionen, die nie ausgeführt werden,
wenn keine Fehler auftreten.

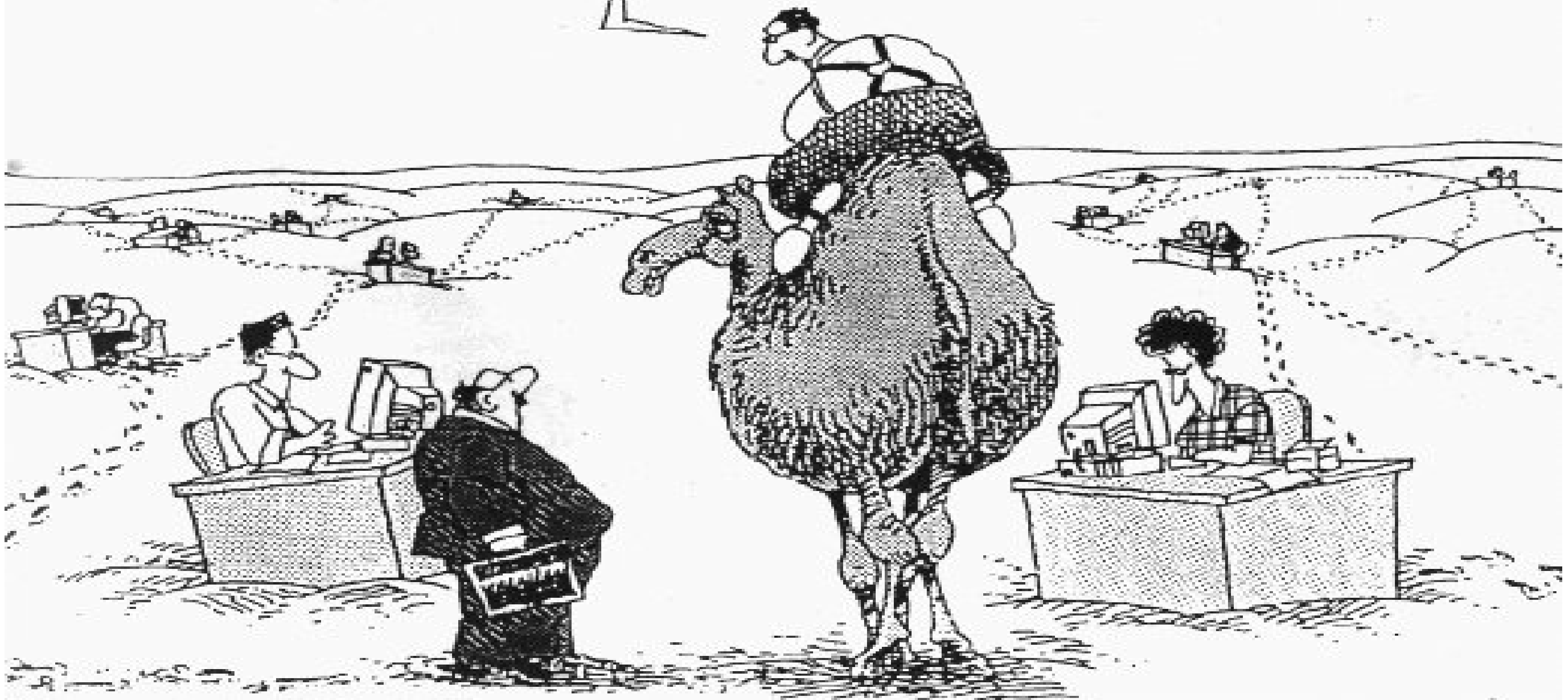
- Zeitredundanz notwendig für Lebendigkeit.
- Einsicht: Korrektoren enthalten Zeitredundanz.
- Einsicht: Zeitredundanz $\Rightarrow$ Speicherredundanz.
- Kompliziert: Genaue Voraussetzungen, unter denen Speicher- und Zeitredundanz notwendig sind. (Details)

Zusammenfassung: Redundanz

fehlertolerant bzgl.	notwendig
Sicherheit	Speicherredundanz
Lebendigkeit	Zeitredundanz + Speicherredundanz

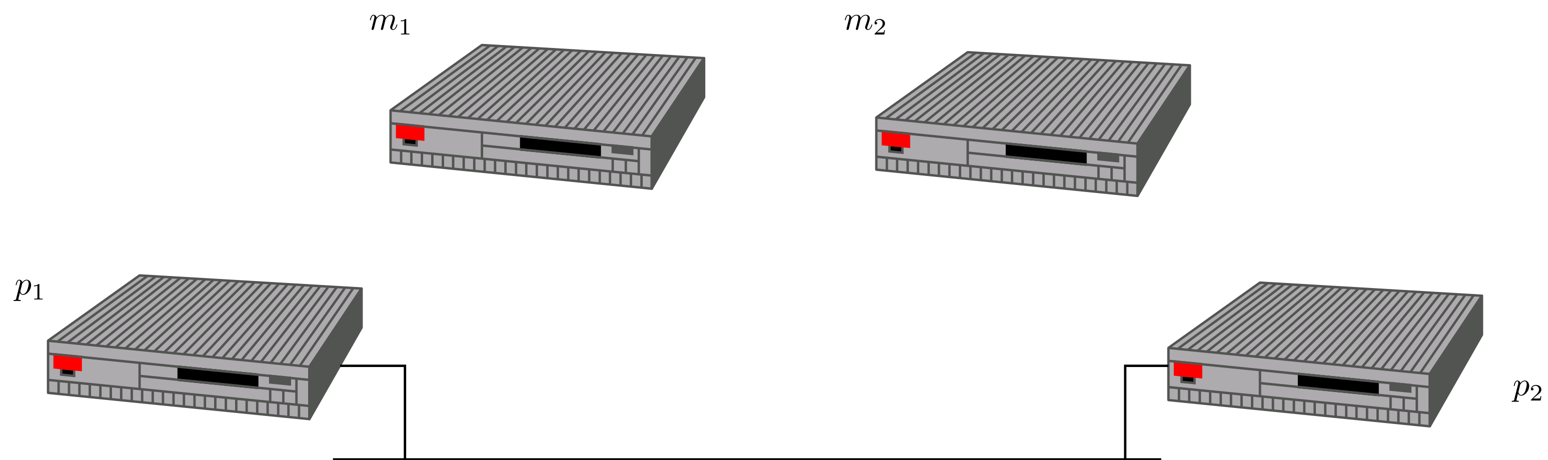
- Beiträge:
  - Erweiterung der Arora/Kulkarni-Theorie um eine Begrifflichkeit der Redundanz.
  - Relation zu *safety* und *liveness* neu!
  - Klärung des unscharfen Begriffs der Zeitredundanz.

Probleme beim Beobachten



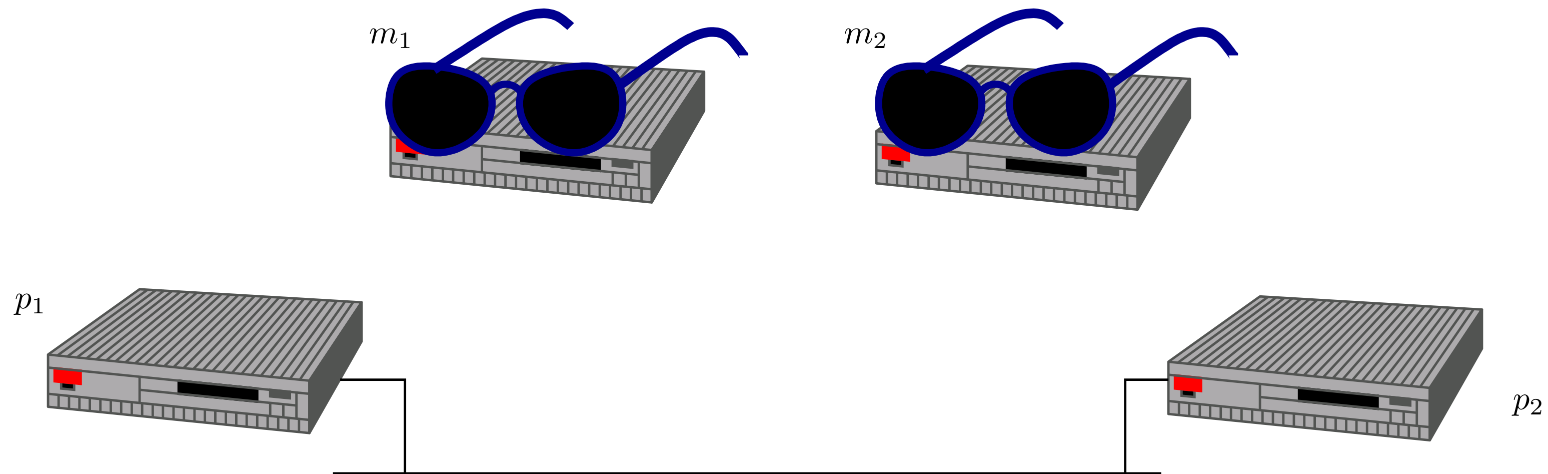
Wie beobachtet man verteilte Systeme?

- Asynchrones System, beobachtete Rechner können abstürzen.



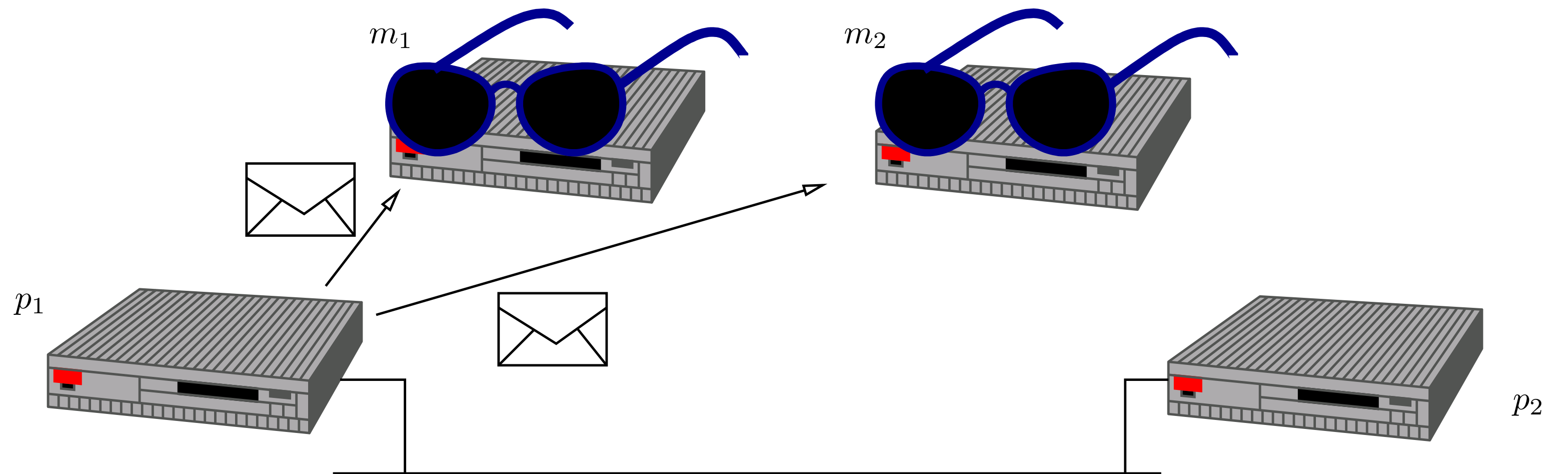
Wie beobachtet man verteilte Systeme?

- Asynchrones System, beobachtete Rechner können abstürzen.



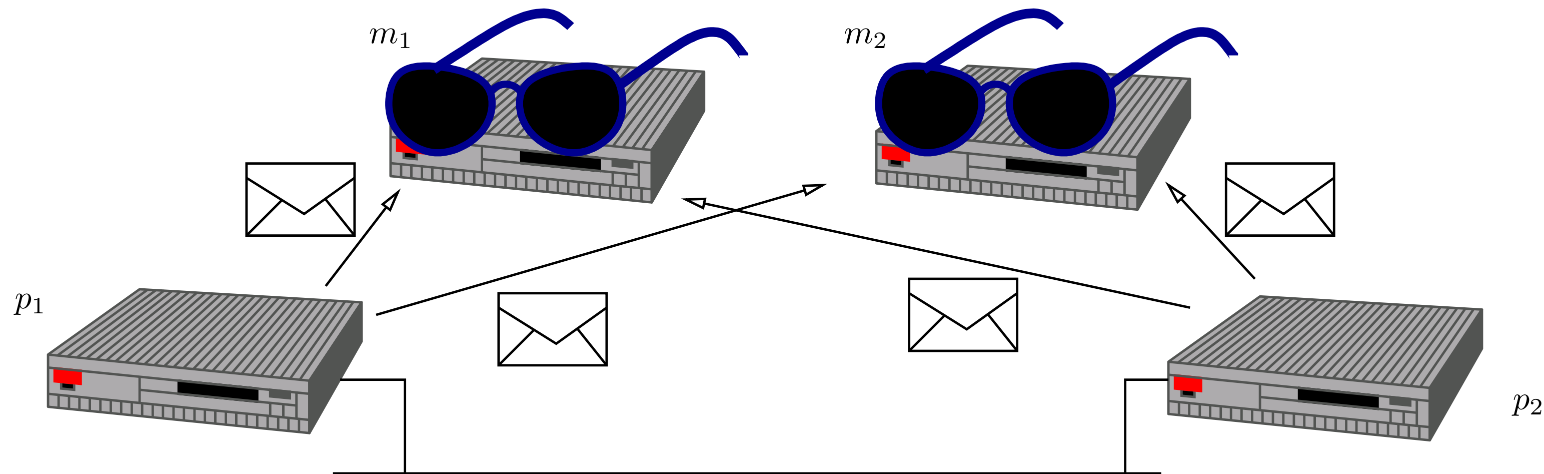
Wie beobachtet man verteilte Systeme?

- Asynchrones System, beobachtete Rechner können abstürzen.

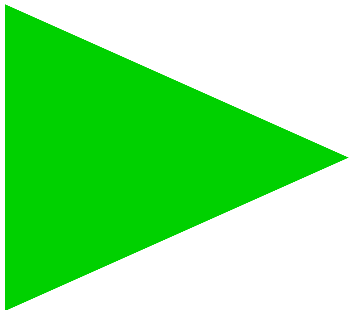


Wie beobachtet man verteilte Systeme?

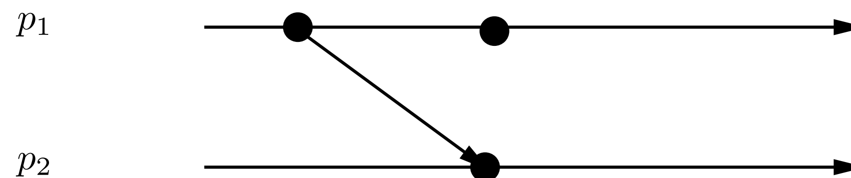
- Asynchrones System, beobachtete Rechner können abstürzen.

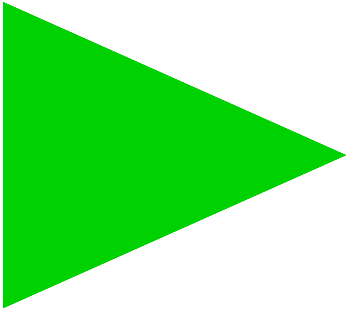


- Keine verläßliche Fehlererkennung.

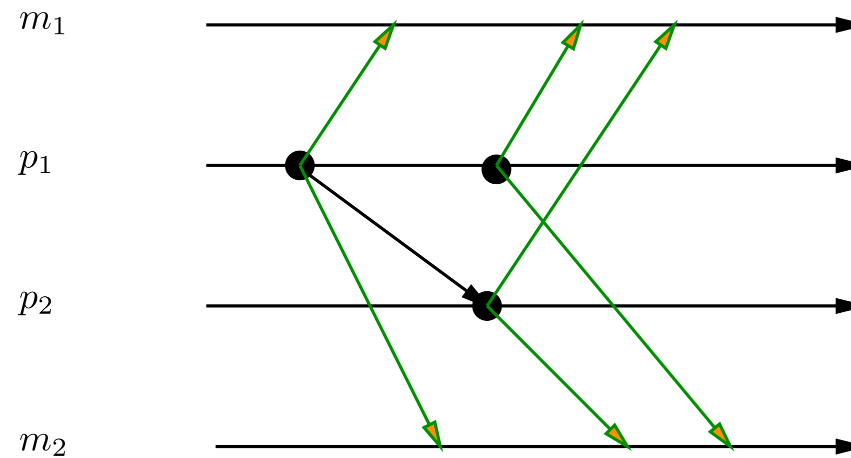


Raum/Zeit-Diagramme

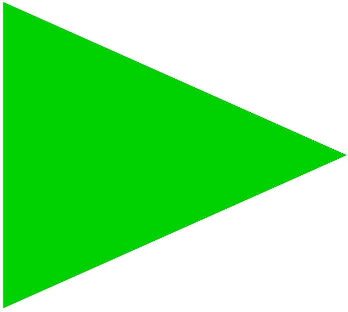




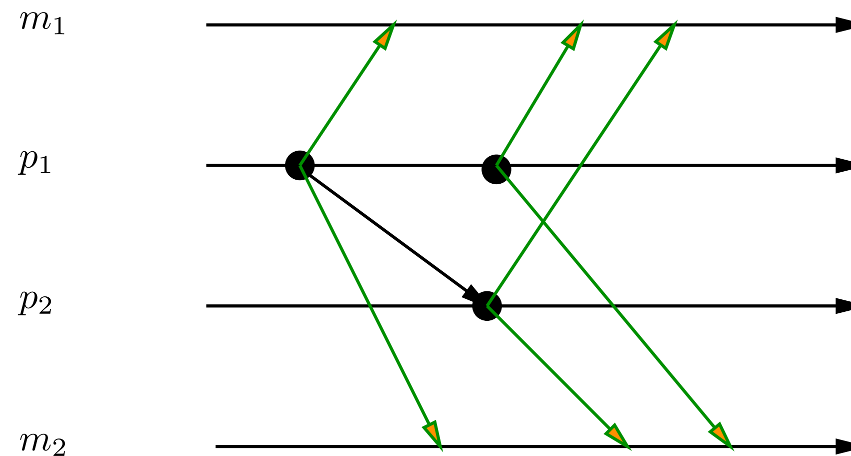
Raum/Zeit-Diagramme



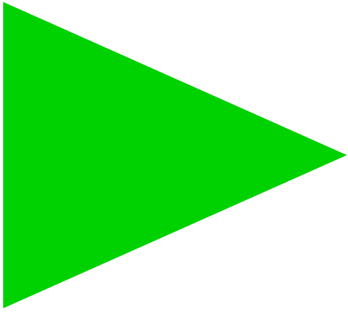
- Beobachten: Erkennen, ob ein globales Prädikat gilt.



Raum/Zeit-Diagramme

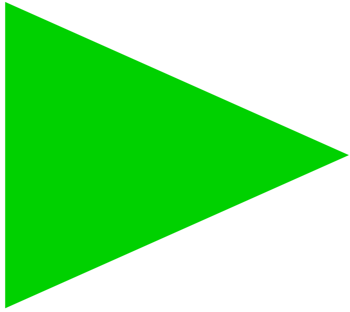


- Beobachten: Erkennen, ob ein globales Prädikat gilt.
- Vorsicht: Beobachterabhängigkeit!



Stand der Forschung

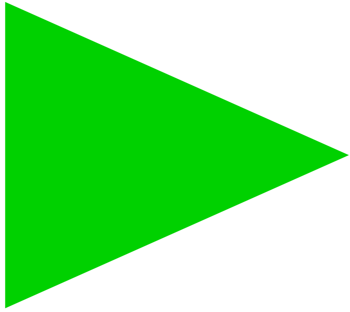
- Ziel: Beobachterunabhängigkeit.
- Beobachtungsmodalitäten
  possibly ($\exists$ eine Beobachtung. . .) und
  definitely ($\forall$ Beobachtungen. . .).
- Ergebnisse gelten nur für fehlerfreie Systeme.
- Arbeit von Garg und Mitchell schränkt Prädikate ein.



Neue Fragestellung

- Beobachter können sich nie sicher sein, ob ein Rechner abgestürzt ist.
- Modalitäten werden wieder beobachterabhängig.

Wie sehen Modalitäten aus, die in fehlerbehafteten Systemen Sinn machen?



Neue Ergebnisse

- Modalität *negotiably*:
  - Pessimistischer Ansatz (*safety*).
  - $\exists$ ein Beobachter, der *possibly* entdeckt? (Erweiterung von *possibly*).
- Modalität *discernibly*:
  - Optimistischer Ansatz (*liveness*).
  - $\forall$ Beobachter gilt *definitely* (Erweiterung von *definitely*).

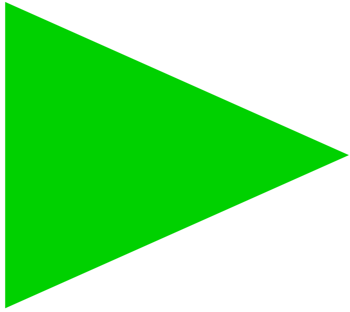
Zusammenfassung (1/2)

- Solide, theoretisch fundierte methodologische Basis notwendig um verläßliche Systeme zu konstruieren.

- Arora/Kulkarni-Theorie erlaubt feingranularen Entwurf von Fehlertoleranzmethoden.

- Neu: Erweiterung der Theorie um Begrifflichkeit der Redundanz.

- Nutzen: Systeme werden bezüglich Redundanz vergleichbar.



Zusammenfassung (2/2)

- Nutzen: Synthese fehlertoleranter Programme auf theoretisch abgesicherter Grundlage möglich.
- Neu: Definition sinnvoller Detektionssemantiken zum Beobachten in fehlerbehafteten Systemen.
- Neu: Entsprechende Beobachtungsalgorithmen.
- Nutzen: Verwendbar als Detektionsmodule in Fehlertoleranzverfahren.

Anhang

- Zusatzfolien zu Redundanz.
- Zusatzfolien zu Fragen der Beobachtung in fehlerbehafteten verteilten Systemen.

Zusatzfolien zu Redundanz

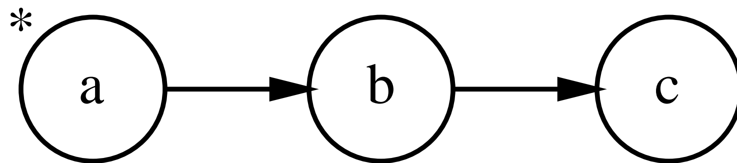
- Definitionen: Fehlermodell, Eigenschaften, *safety*, *liveness*, fehlertolerante Version, Lebendigkeitsannahme, Fusionsabgeschlossenheit.

- Theoreme: Fehlertoleranz und Sicherheit bzw. Lebendigkeit.

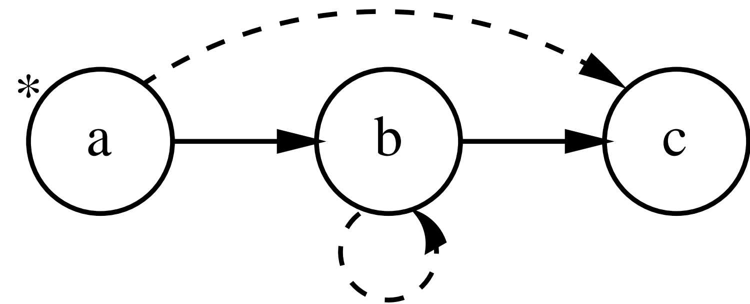
- Beispiel: TMR.

Definition: Fehlermodell

- Fehlermodell = Transformation F von Automaten, die Zustandsübergänge einbaut.



Σ



$F(\Sigma)$

Eigenschaften

- Automaten sind Generatoren von *Abläufen*.
- Ablauf $=$ Folge $s_1, s_2, \ldots$ von Zuständen.
- Eigenschaft $=$ Menge $\{\sigma_1, \sigma_2, \ldots\}$ von Abläufen.
- Ein Automat besitzt Eigenschaft E wenn alle seine Abläufe in E liegen.

Sicherheit und Lebendigkeit

- Sicherheitseigenschaft (*safety*): Eigenschaft, die immer im Endlichen verletzt wird.
- Beispiel: wechselseitiger Ausschluß.
- Lebendigkeitseigenschaft (*liveness*): Eigenschaft, die nur im Unendlichen verletzt werden kann.
- Beispiel: Aushungerungsfreiheit.
- Sicherheit und Lebendigkeit sind fundamental [1].

Fehlertolerante Versionen

- Designprozeß:
  - Man hat ein System Σ_1, welches Sicherheitseigenschaft S verletzt, wenn Fehler aus F auftreten.
  - Möchte Σ_1 gerne in Σ_2 umwandeln, so daß Σ_2 S erfüllt, auch wenn Fehler aus F auftreten.
  - Σ_2 soll aber im fehlerfreien Fall dasselbe Verhalten haben wie Σ_1.

Σ_2 ist die *fehlertolerante Version* von Σ_1.

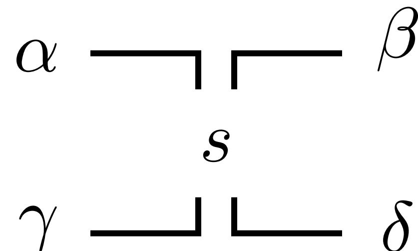
Fehlertoleranz und Sicherheit

- “Nie x” ist eine Sicherheitseigenschaft.
- Wann kann man fehlertolerante Versionen bezüglich einem Fehlermodell F und einer Sicherheitseigenschaft S bauen?
- Unmöglich, falls S direkt mittels Transitionen aus F verletzt werden kann.
- Oft möglich durch Löschen redundanter Transitionen.
- Resultierendes System Σ_2 hat Speicherredundanz.

Sicherheit und Speicherredundanz

- Speicherredundanz notwendig, um fehlertolerant bezüglich einer Sicherheitseigenschaft zu werden.
- Voraussetzung: F muß in Σ_1 und Σ_2 dieselben Fehler einbauen.
- Annahme: Sicherheitseigenschaft ist *fusionsabgeschlossen*.

Fusionsabgeschlossenheit



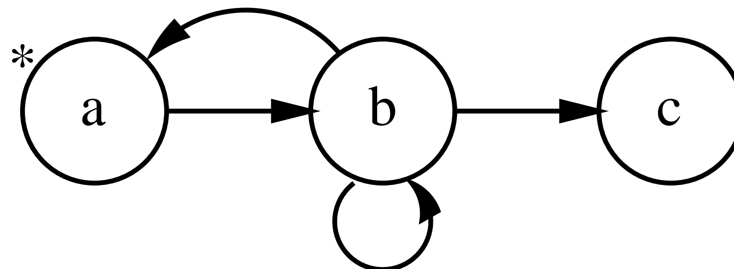
- Sicherheit allgemein: Ausschluß einer Menge endlicher Präfixe.
- Fusionsabgeschlossene Sicherheit: Ausschluß einer Menge von Transitionen.
- “Man kann an der Transition erkennen, ob sie ausgeführt werden darf oder nicht.”

Fehlertoleranz und Lebendigkeit

- “Immer wieder x” ist eine Lebendigkeitseigenschaft.
- Was muß man tun, um fehlertolerante Versionen bezüglich einer Lebendigkeitseigenschaft zu konstruieren?

Lebendigkeitsannahmen

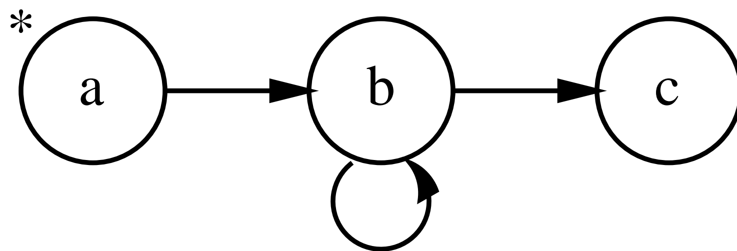
- Automat $\Sigma = (C, I, T, A)$



- Maximalität
- Schwache Fairness

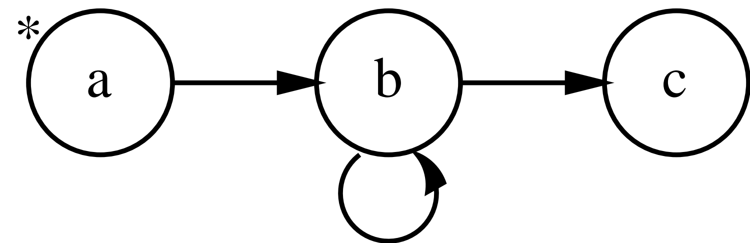
Fehlermodelle und Lebendigkeit

- Fehlermodelle können auch die Lebendigkeitsannahme abschwächen.
- Beispiel: Abschwächung von schwacher Fairness zu Maximalität.



Σ

A = schwache Fairness



$F(\Sigma)$

A = Maximalität

Fehlertoleranz und Lebendigkeit

- Wann kann man fehlertolerante Versionen bezüglich einem Fehlermodell F und einer Lebendigkeitseigenschaft L bauen?
- Unmöglich, falls aus direktem Programmfluß ein *livelock* ausschließlich in F-Transitionen passieren kann.
- Oft möglich durch Hinzufügung von redundanten Transitionen.
- Resultierendes System Σ_2 hat Zeitredundanz.

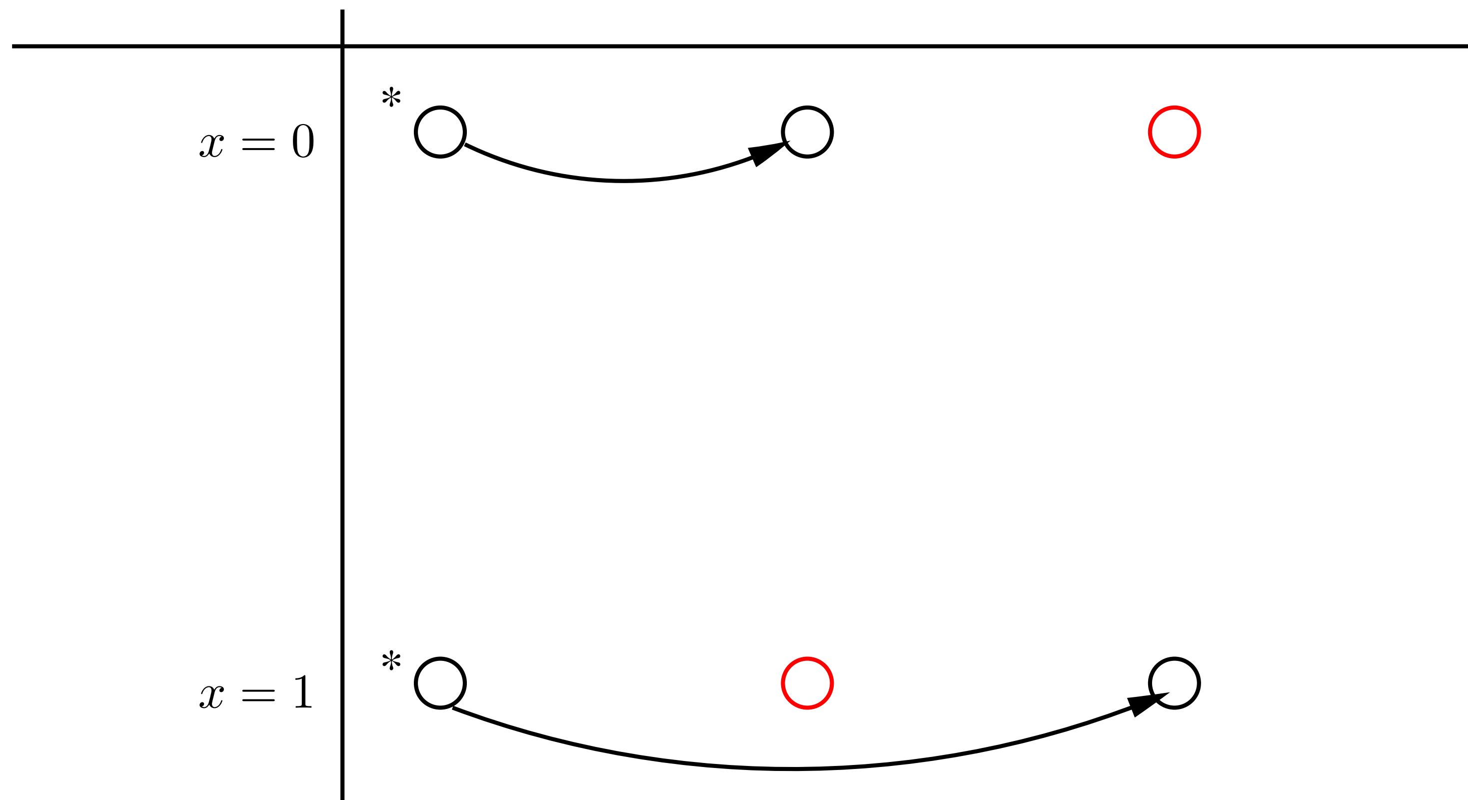
Lebendigkeit und Zeitredundanz

- Zeitredundanz notwendig, um fehlertolerant bezüglich einer Lebendigkeitsspezifikation zu werden.
- Voraussetzung: F schwächt bei beiden Programmen die Lebendigkeitsannahme in derselben Art ab.

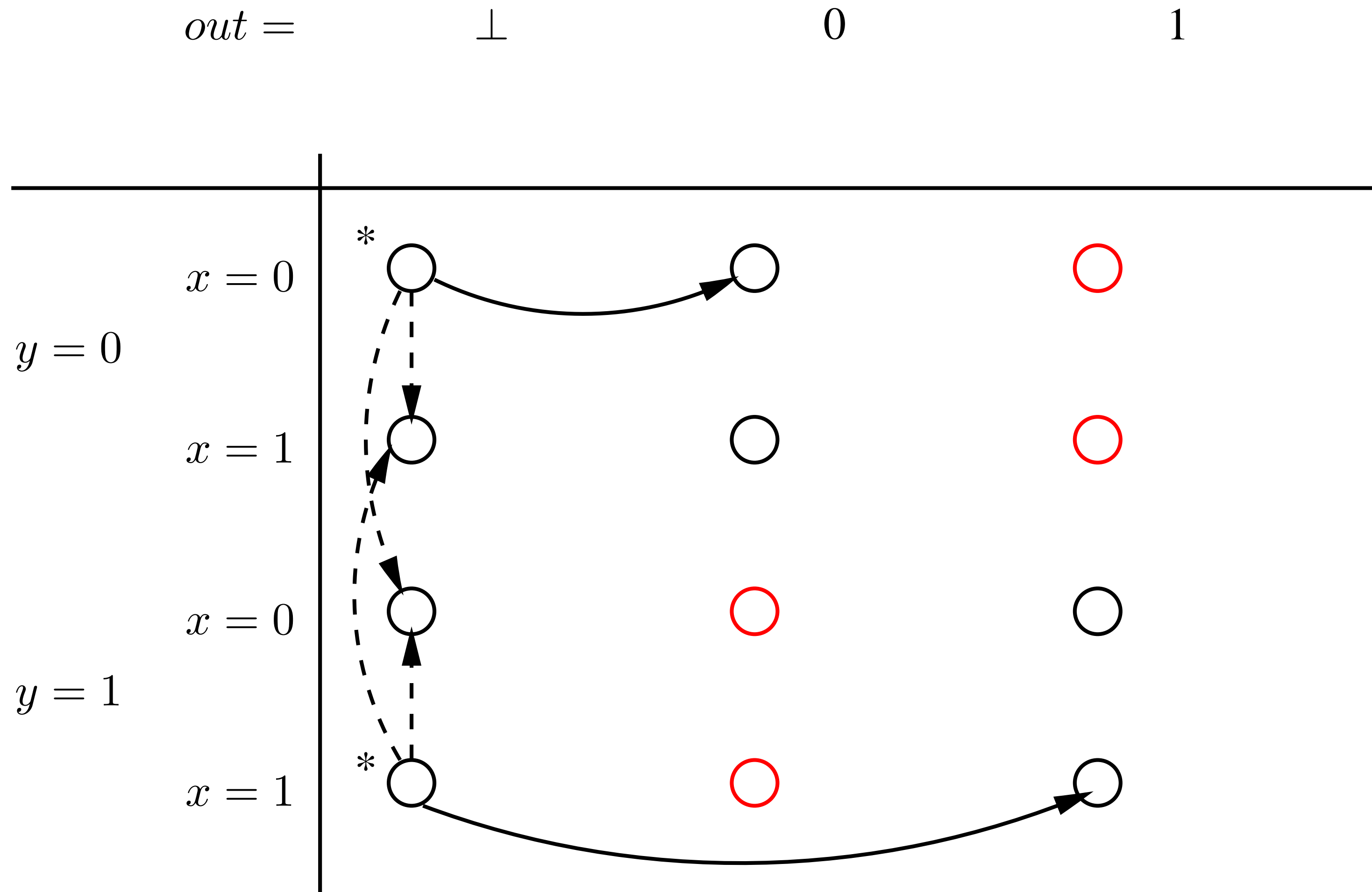
Beispiel: TMR

- Komponente x berechnet einen Wert aus $\{0, 1\}$.
- Spezifikation: Nur korrekter Wert soll auf Ausgang out geschrieben werden (Sicherheit) und schließlich soll out von $\perp$ auf 0 oder 1 wechseln (Lebendigkeit).
- Zusätzliche Komponenten y und z stehen zur Verfügung.
- Fehlermodell: maximal eine Komponente berechner fehlerhaften Wert.

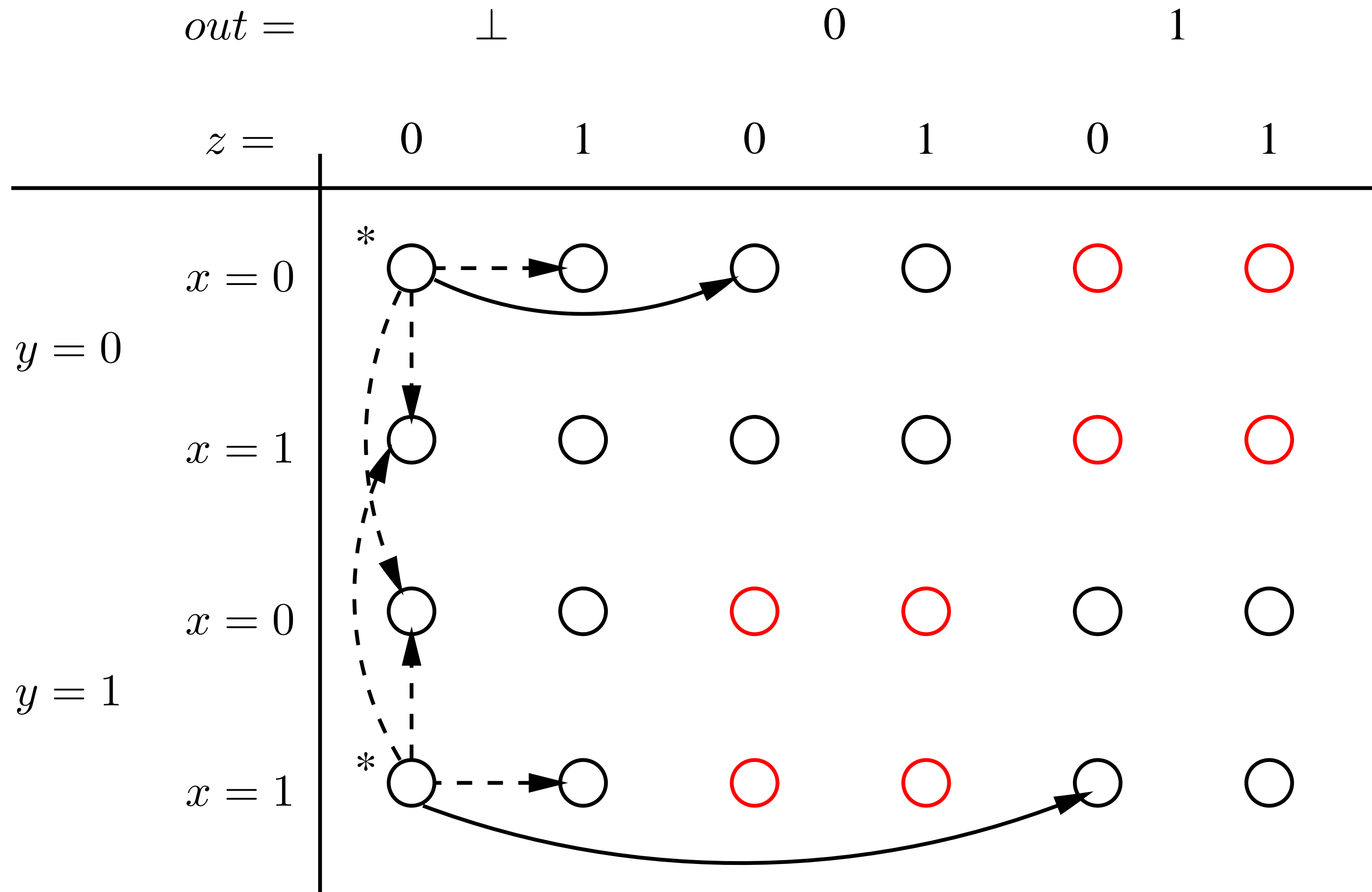
Speicher- und Zeitredundanz in TMR



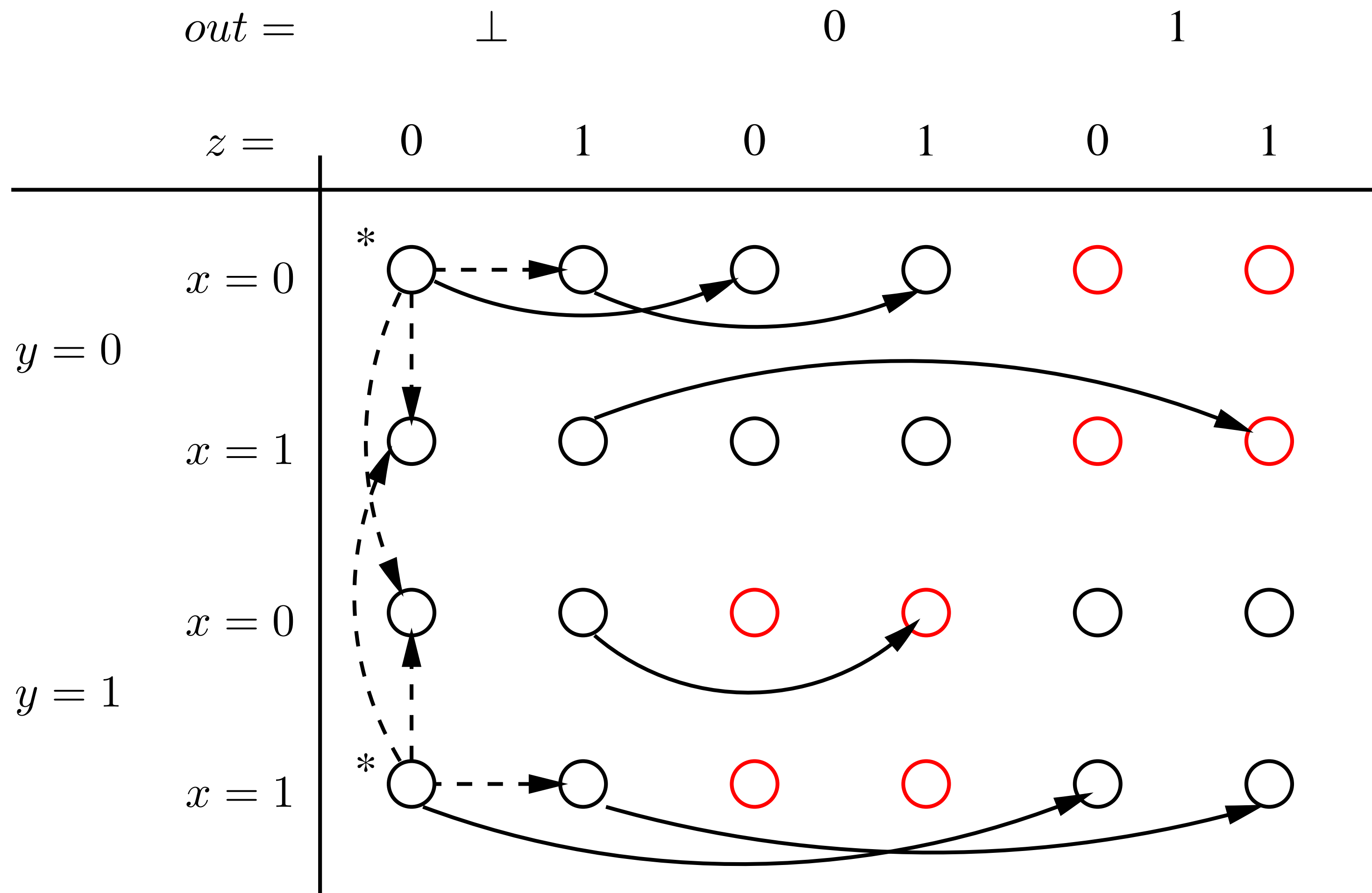
Speicher- und Zeitredundanz in TMR



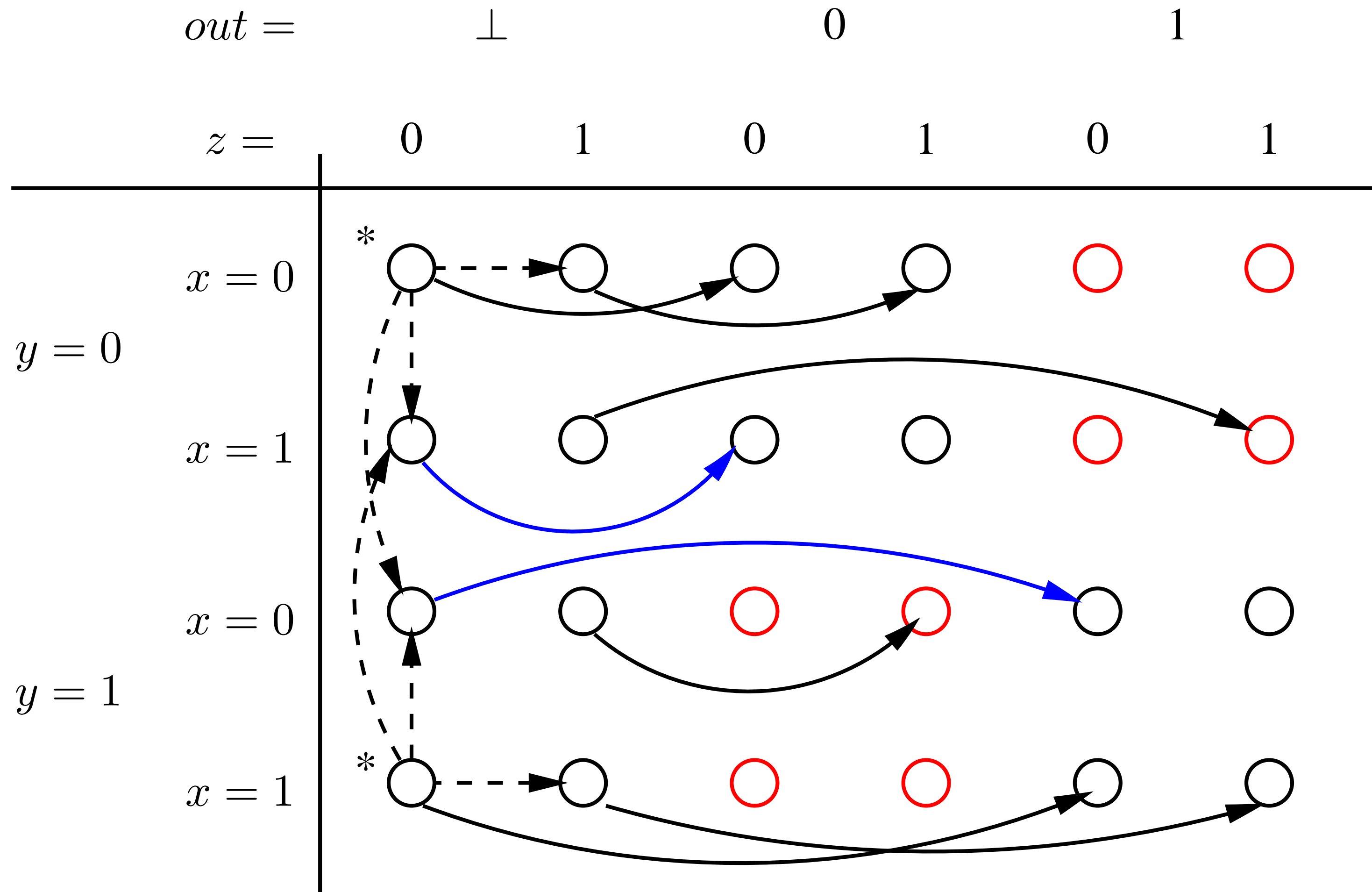
Speicher- und Zeitredundanz in TMR



Speicher- und Zeitredundanz in TMR



Speicher- und Zeitredundanz in TMR



Zusatzfolien zu Fragen der Beobachtung in fehlerbehafteten verteilten Systemen

- Typen von Prädikaten.
- Unzuverlässige Fehlerdetektion.
- Erweiterter Zustandsraum.
- Neue Modalitäten.
- Entdeckungsalgorithmen.
- Weitergehende Fragen.

Predicate detection in faulty asynchronous systems

- crash fault assumption = at most t processes simply stop executing steps.
- For the moment: restrict crash faults to application processes only (monitors always stay alive).
- Predicate up_i refers to functional state of p_i.

New types of predicates

- Can be used in predicates:
  - Process p_i crashed after 4th event: $\neg up_i \wedge ec_i = 4$
  - Every process either commits or crashes: $\forall i : \neg up_i \vee commit_i$
- Idea: find suitable analogies to *possibly* and *definitely* for these types of predicates.

Implementable failure detection (1/2)

- Every monitor must keep up_i up to date (failure detection).,
- Can ensure eventual detection, but cannot avoid false suspicions.
- Terminology: failure detectors *suspect* and *rehabilitate* application processes.

Implementable failure detection (2/2)

- Best we can do: a non-crashing process is not permanently suspected [6].

- For observation purposes: add causality information to suspicions:
  - "m_j suspects p_i after event e_k on p_i."
  - "m_j rehabilitates p_i after event e_k on p_i."

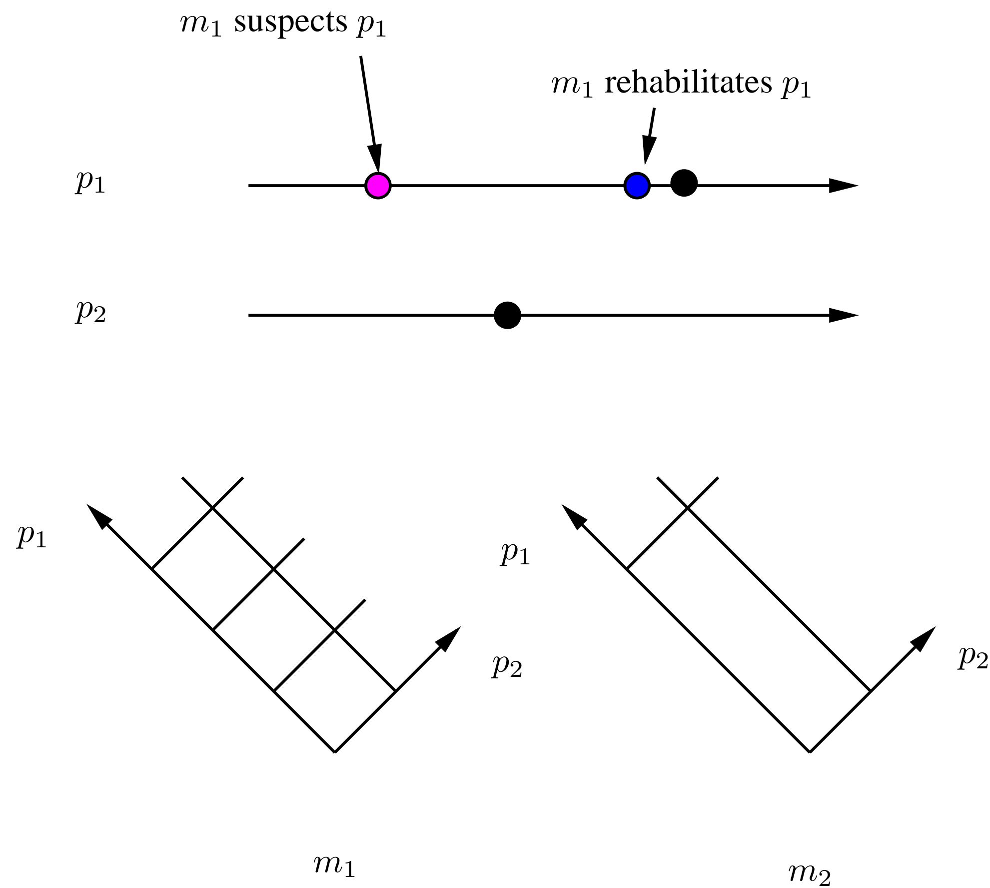
- Assume: between two events at most one suspicion and rehabilitation.

Lattice over extended state space

- Treat up_i as a variable on p_i.
- Suspicion/rehabilitation is a simple state change of p_i (extended state space).
- Change of up in consistent states yields again consistent states.
- Integration of suspicions/rehabilitations into state lattice yields new lattice (over extended state space).
- Use this lattice for predicate detection.

New observer dependence

- *possibly*/*definitely* not observer-invariant.



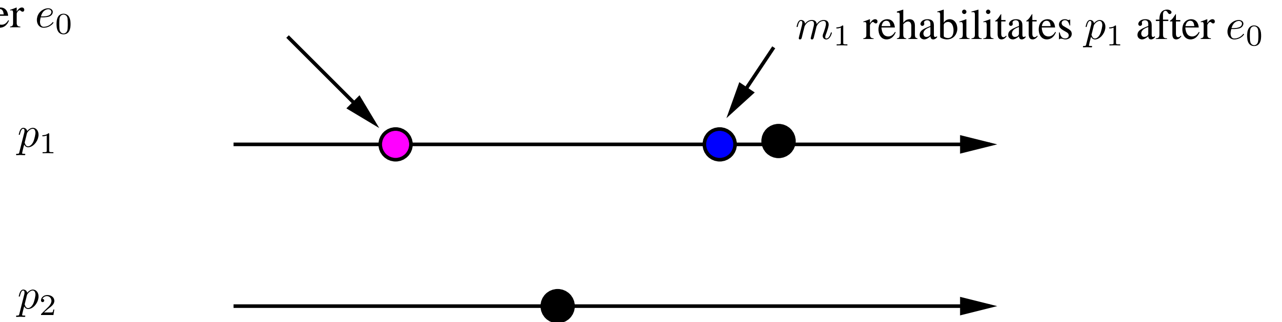
Global failure detector semantics

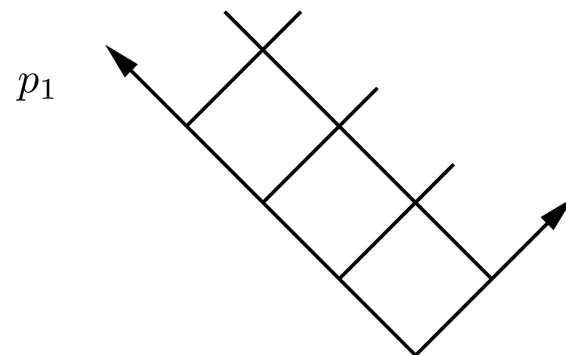
- Problem: false suspicions.
- Solution: define “global” failure detector semantics.
- p_i is (globally) suspected after e_k iff ...
  - (pessimistic) $\exists$ a monitor which suspects p_i after e_k.
  - (optimistic) $\forall$ monitors suspect p_i after e_k.
- Can define pessimistic and optimistic state lattice (union and intersection of all monitor lattices).

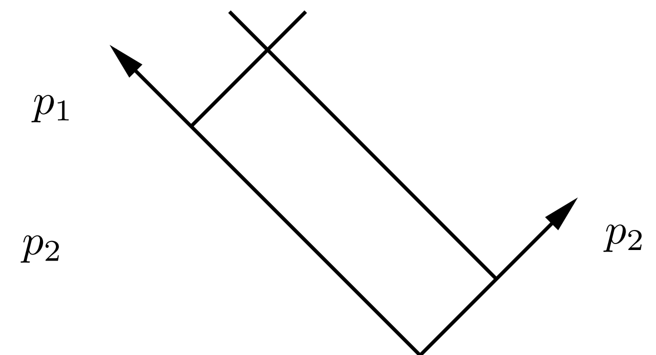
New modalities

- Given predicate φ on extended state space.
- $negotiably(\varphi)$ holds iff $possibly(\varphi)$ holds on pessimistic state lattice.
- $discernibly(\varphi)$ holds iff $definitely(\varphi)$ holds on optimistic state lattice.

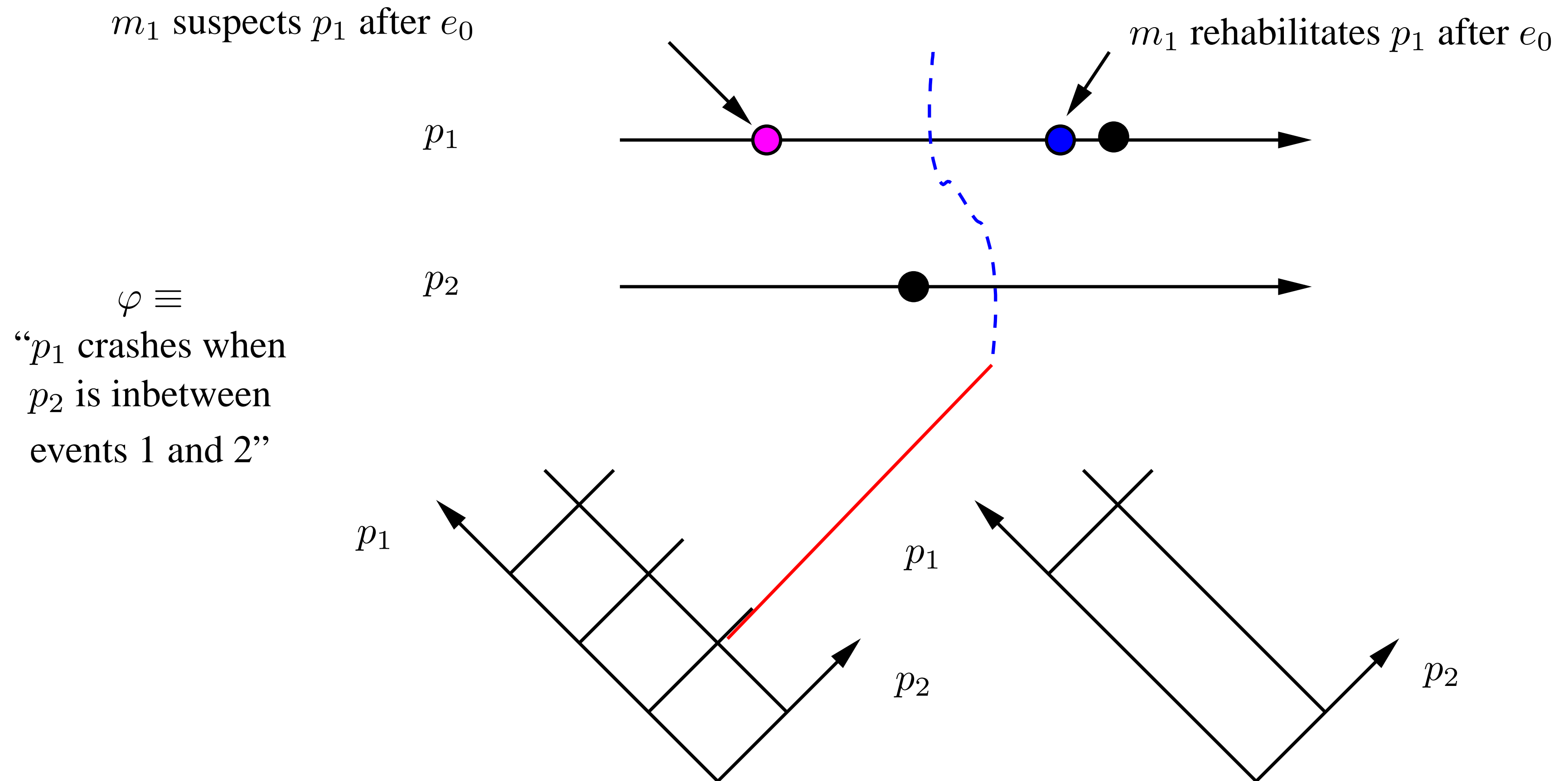
New modalities example



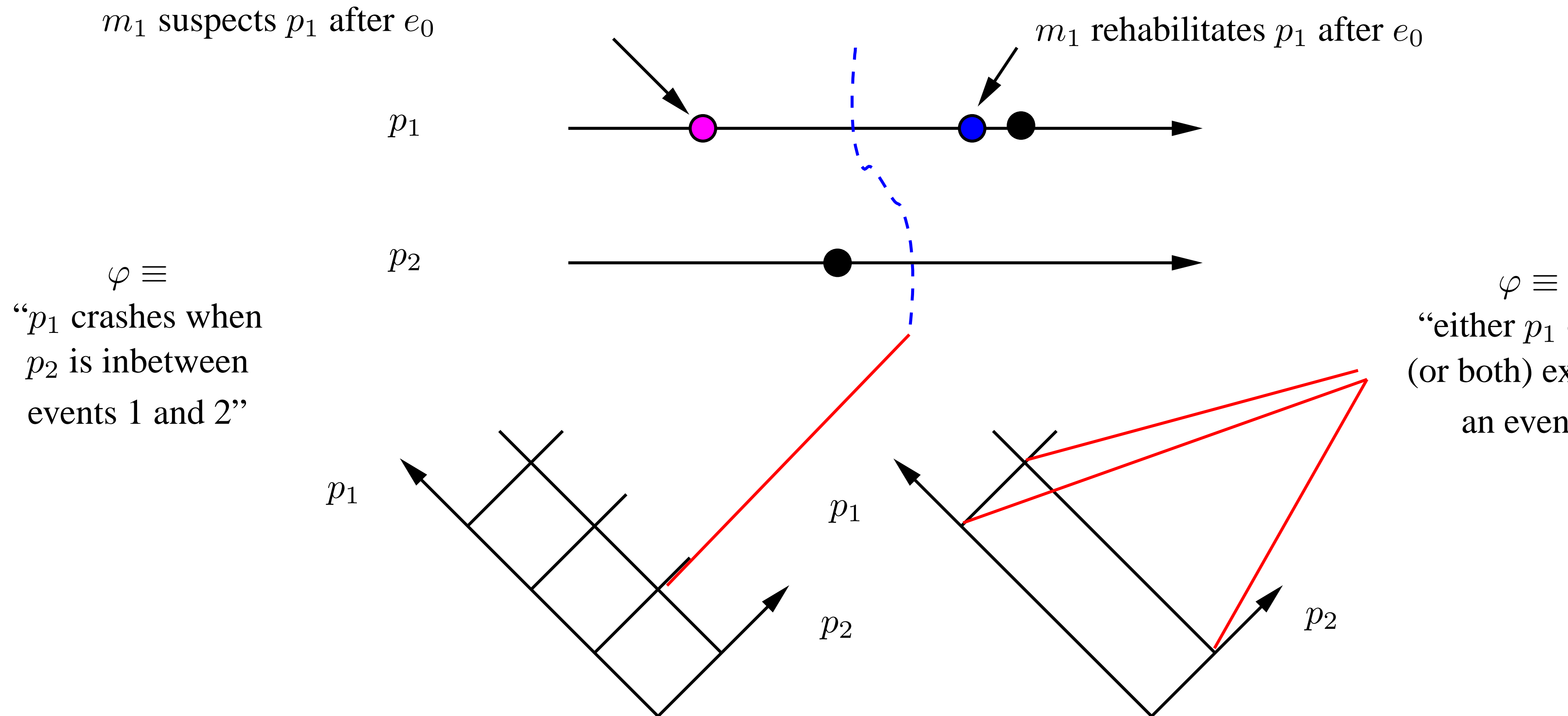




New modalities example



New modalities example



Intuition behind new modalities (1/2)

- Optimistic/pessimistic lattice can be understood in analogy to optimistic/pessimistic network protocols:
  - pessimistic: be careful all the time, take immediate action if something bad has possibly happened.

$\Rightarrow$ use *negotiably* to trigger action.
  - optimistic: go ahead without synchronization and hope for the best, deal with conflicts only when necessary.

$\Rightarrow$ use *discernibly* to ignore spurious suspicions.

Intuition behind new modalities (2/2)

- Understandable in analogy to $possibly/definitely$:
  - Safety requirement $\Box\varphi$: take action if $negotiably(\neg\varphi)$ is detected.
  - Liveness requirement $\Diamond\varphi$: validated if $discernibly(\varphi)$ is detected.

Overview of results

- Observation modalities *negotiably* and *discernibly*. . .
  - do not solve all problems in crash-affected systems.
  - reflect by their definition the inherent problem of crash failure detection.
  - can be understood in analogy to *possibly* and *definitely*.
  - can be detected in asynchronous systems, even if monitors may crash.

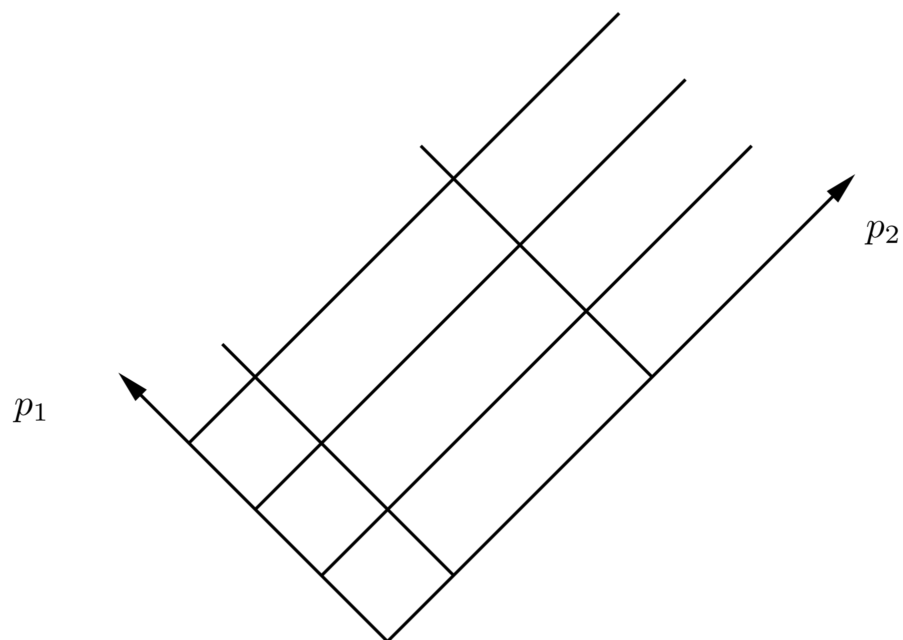
Detection algorithms in a nutshell (1/2)

- Let monitors causally broadcast their suspicions to all other monitors.
- Eventually all monitor lattices converge.
- Can then do *possibly*/*definitely* detection in observer invariant state lattices (use standard algorithms).
- Problem: how know that there will be no “late” failure detector events arriving?

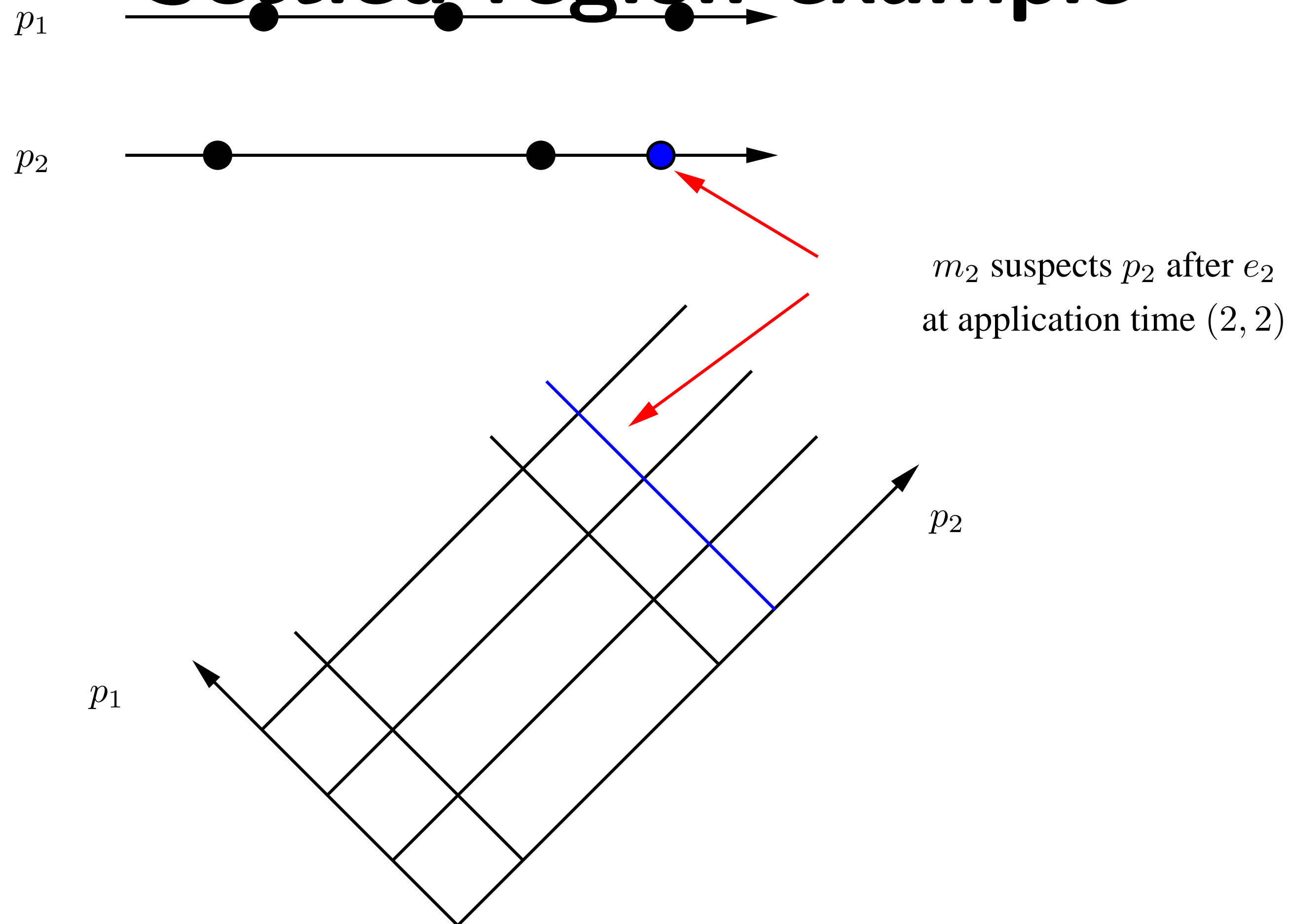
Detection algorithms in a nutshell (2/2)

- Solution:
  - Monitors piggyback coordinates of most recent global state they have seen: per monitor stable region.
  - Take intersection of all monitor regions: globally settled region.
  - Steadily expand settled region, extract optimistic/pessimistic data and do *possibly*/*definitely* detection on it.

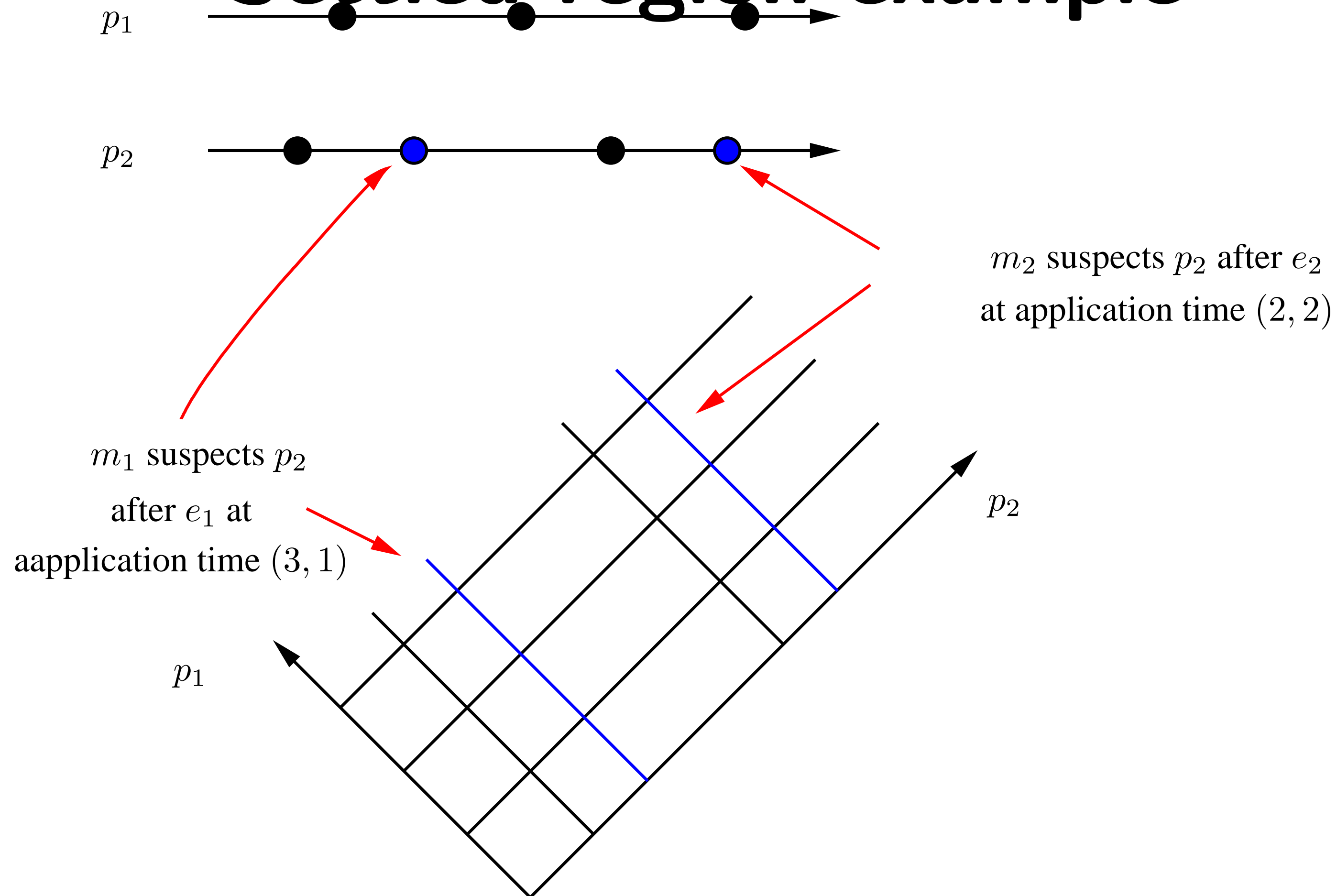
Settled region example



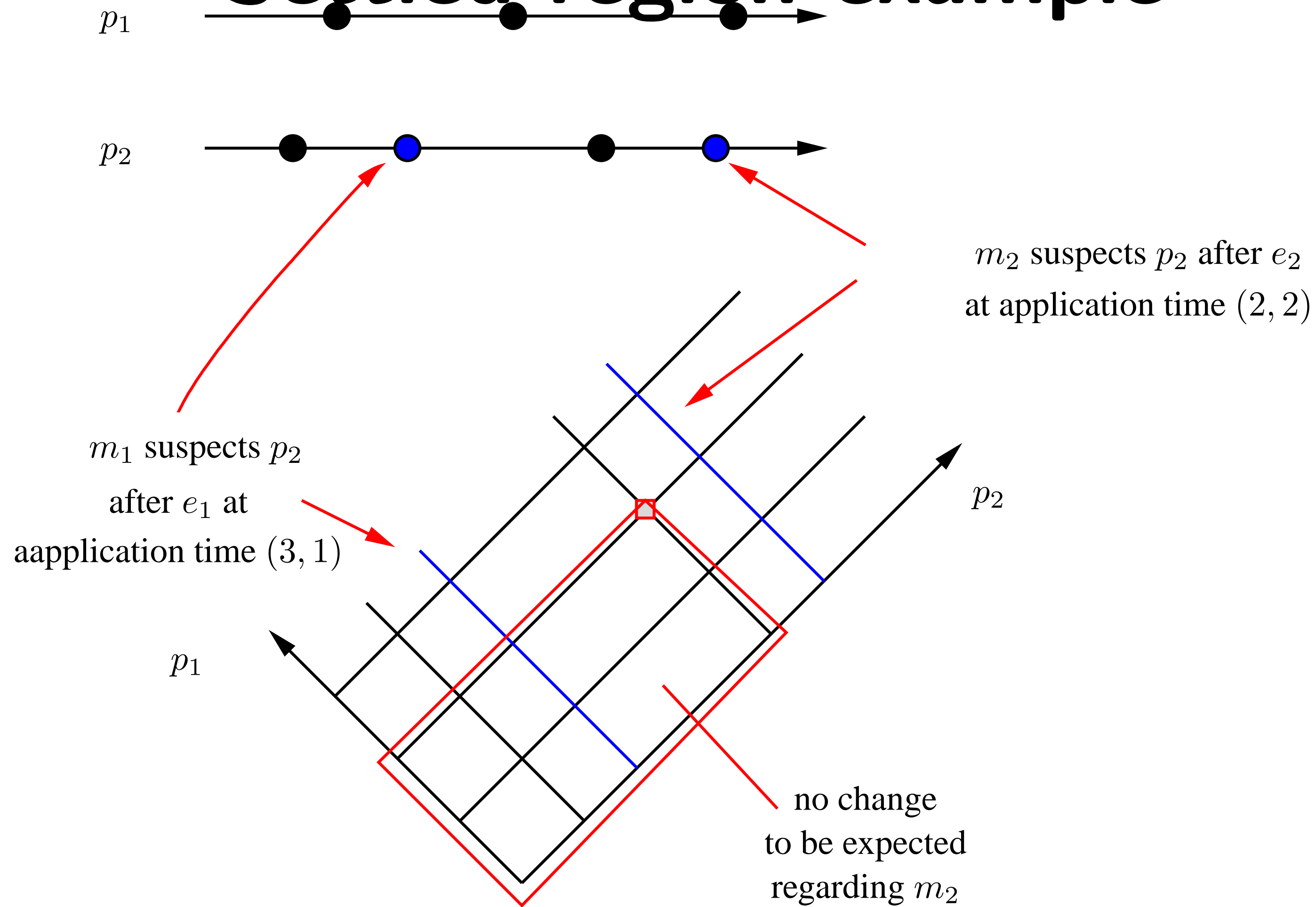
Settled region example



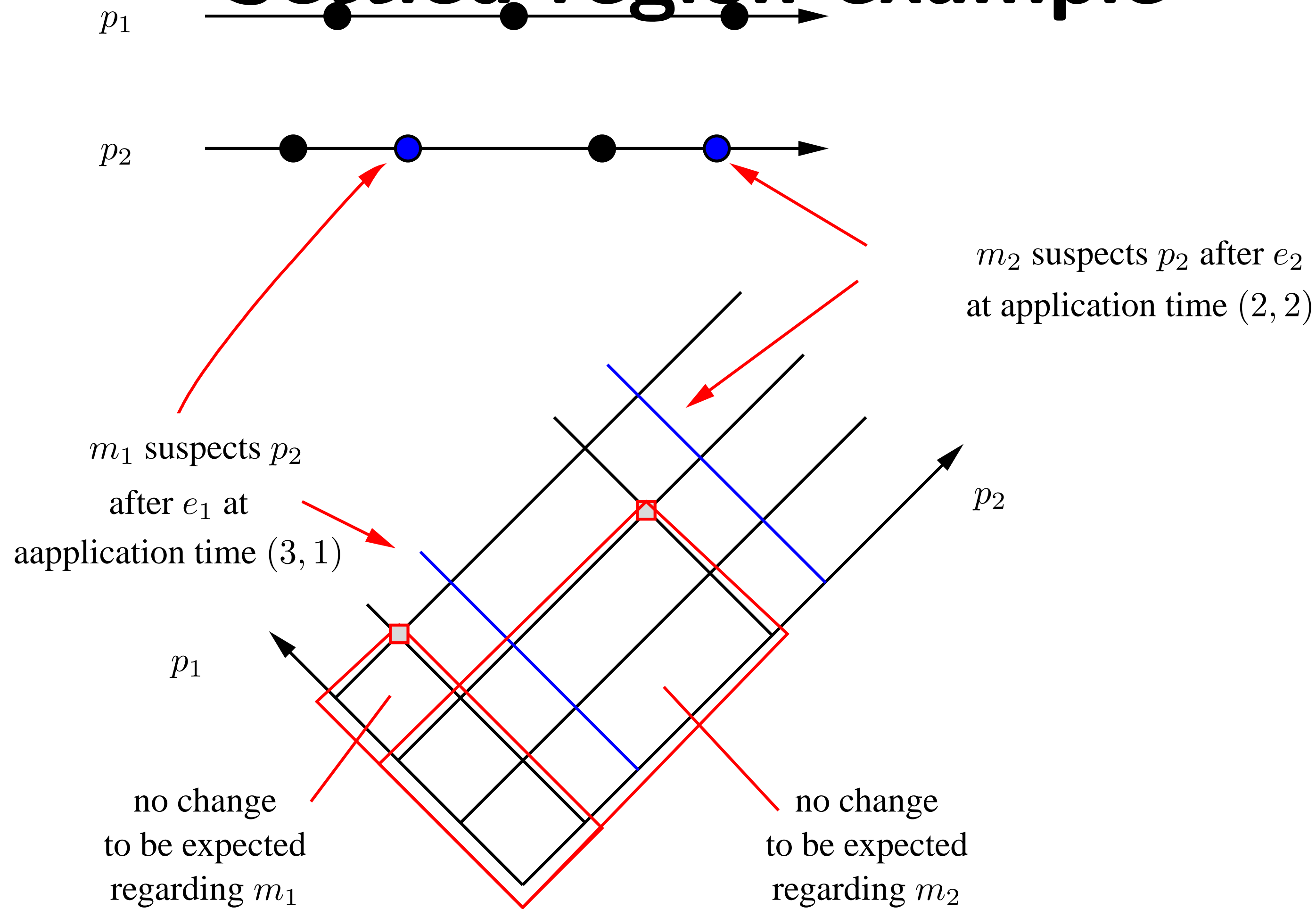
Settled region example



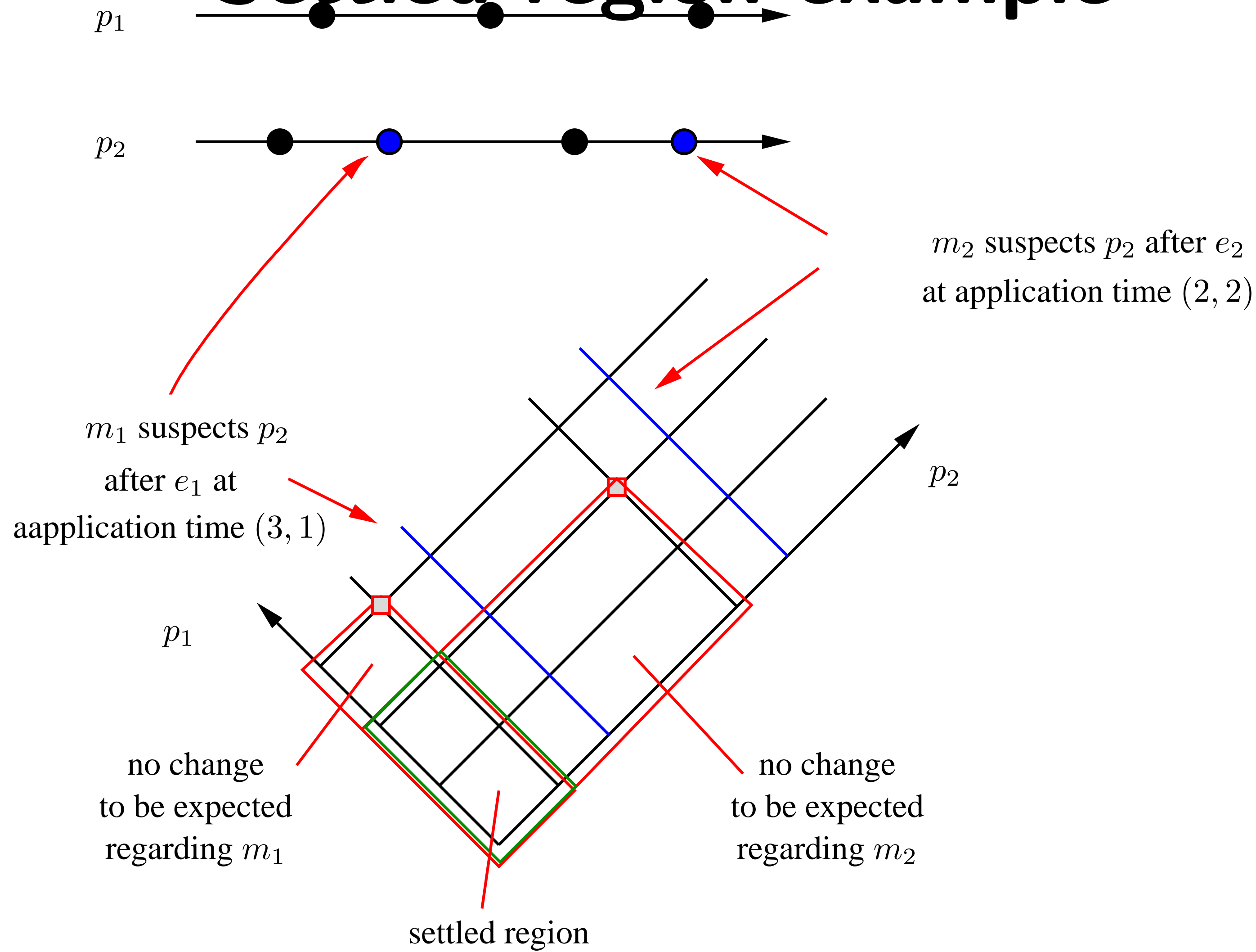
Settled region example



Settled region example



Settled region example



Advanced topics

- Algorithm works under assumption that no monitors fail.
- If monitors can fail, detection becomes harder:
  - Can still detect *negotiably* without a stable region.
  - Detection *discernibly* impossible, because accurate failure detection is needed.
  - A weaker variant (*t-discernably*) can be detected at the price of having a majority of correct monitors.

Complexity

- Complexity:
  - general predicate detection is NP-complete [3].
  - Our detection algorithms are only wrappers around possibility/definitely detection.
  - Study restricted classes of predicates.

Restricted predicates

- Perfect failure detectors available:
  - No false suspicions.
  - Optimistic/pessimistic lattice are the same.
- Perfect failure detectors and crash predicates:
  - Predicates are stable.
  - $possibly = definitely \rightarrow negotiably = discernibly$

Danksagungen

- Diese Folien wurden hergestellt unter Verwendung von pdfLaTeX und Klaus Guntermanns PPower4.

Literatur

[1] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.

[2] Anish Arora and Sandeep S. Kulkarni. Component based design of multitolerant systems. *IEEE Transactions on Software Engineering*, 24(1):63–78, January 1998.

[3] Craig M. Chase and Vijay K. Garg. Detection of global predicates: Techniques and their limitations. *Distributed Computing*, 11(4):191–201, 1998.

[4] Robert Cooper and Keith Marzullo. Consistent detection of global predicates. *ACM SIGPLAN Notices*, 26(12):167–174, December 1991.

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

[6] Vijay K. Garg and J. Roger Mitchell. Implementable failure detectors in asynchronous systems. In *Proc. 18th Conference on Foundations of Software Technology and Theoretical Computer Science*, number 1530 in Lecture Notes in Computer Science, Chennai, India, December 1998. Springer-Verlag.

[7] Sandeep S. Kulkarni and Anish Arora. Automating the addition of fault-tolerance. In Mathai Joseph, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems, 6th International Symposium (FTRTFT 2000) Proceedings*, number 1926 in Lecture Notes in Computer Science, pages 82–93, Pune, India, September 2000. Springer-Verlag.

[8] Keith Marzullo and Gil Neiger. Detection of global state predicates. In *Proceedings of the 5th International Workshop on Distributed Algorithms (WDAG91)*, pages 254–272, 1991.

[9] T. R. N. Rao and E. Fujiwara. *Error-control coding for computer systems*. Prentice-Hall, 1989.