Modular Synthesis of FDIR Recovery Strategies from Fault Trees

Joint ongoing work with S. Müller, A. Jordon and A. Gerndt
(DLR German Aerospace Center, Braunschweig)

Thomas Noll (noll@cs.rwth-aachen.de)

MOVES Söllerhaus Workshop
February 1–8, 2020

Software Modeling and Verification Chair
RWTH AACHEN UNIVERSITY

Content

Fault Detection, Isolation and Recovery

FDIR

Even well designed systems cannot exclude the occurrence of faults

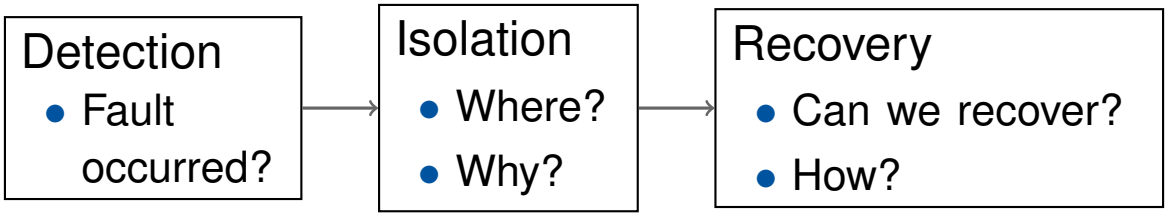
- But not every fault is a failure
- FDIR tries to prevent faults from turning into (system-level) failures

Fault Detection, Isolation and Recovery

FDIR

Even well designed systems cannot exclude the occurrence of faults

- But not every fault is a failure
- FDIR tries to prevent faults from turning into (system-level) failures

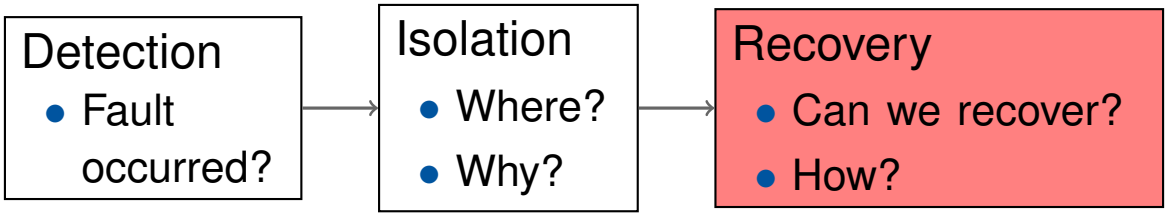


Fault Detection, Isolation and Recovery

FDIR

Even well designed systems cannot exclude the occurrence of faults

- But not every fault is a failure
- FDIR tries to prevent faults from turning into (system-level) failures



"Failure is a bruise. Not a tattoo." – Jon Sinclair

Problem statement

Given a (dynamic) fault tree, determine an (efficiently representable) recovery strategy that maximises the overall reliability of the system.

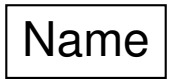
Content

Static Fault Trees

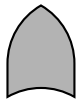
Fault trees

Fault trees describe how faults starting from basic error events propagate through components and sub-systems and eventually lead to a top-level failure.

Basic Event



Intermediate event



OR

AND

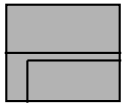
k-VOTE

Dynamic Fault Trees

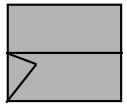
Dynamic fault trees

Dynamic fault trees extend static fault trees with temporal dependencies and spare management.

PAND



SPARE



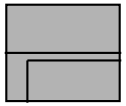
FDEP

Dynamic Fault Trees

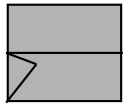
Dynamic fault trees

Dynamic fault trees extend static fault trees with temporal dependencies and spare management.

PAND



SPARE

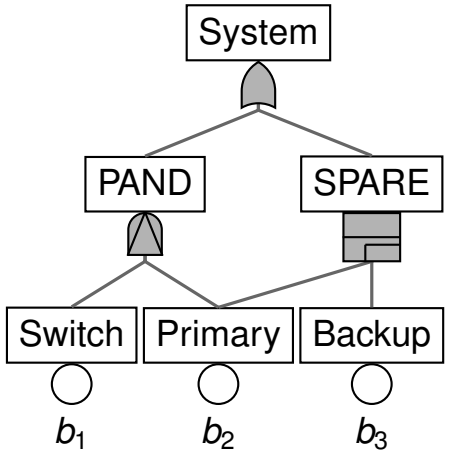


FDEP

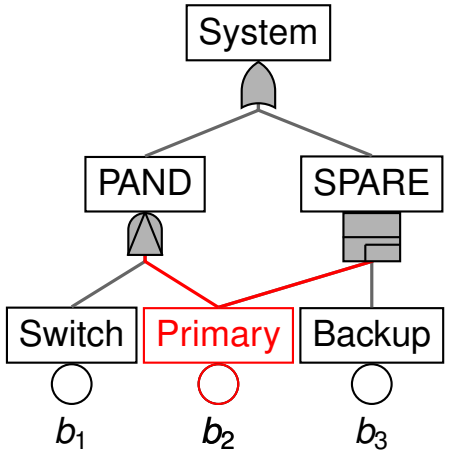
Deterministic semantics of SPARE gates

Spares are claimed whenever the primary fails, and always from left to right.

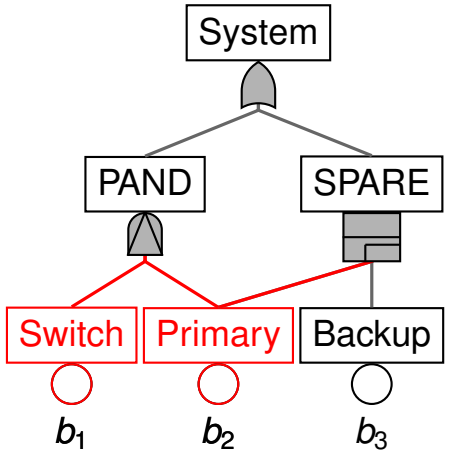
Fault Tree Examples and Challenges



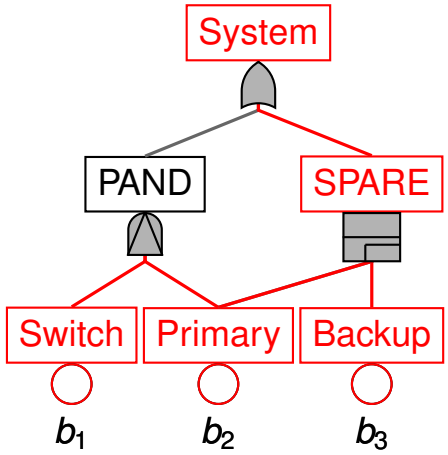
Fault Tree Examples and Challenges



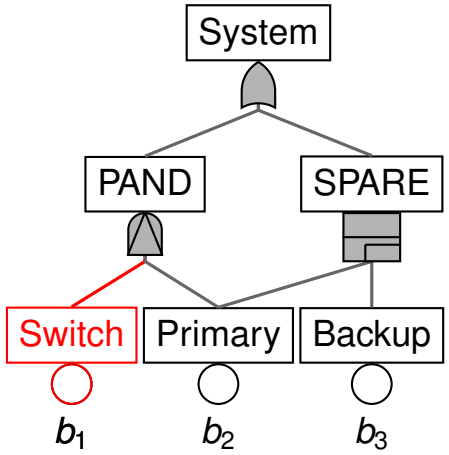
Fault Tree Examples and Challenges



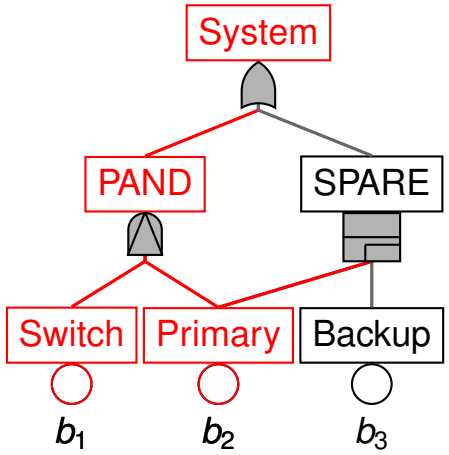
Fault Tree Examples and Challenges



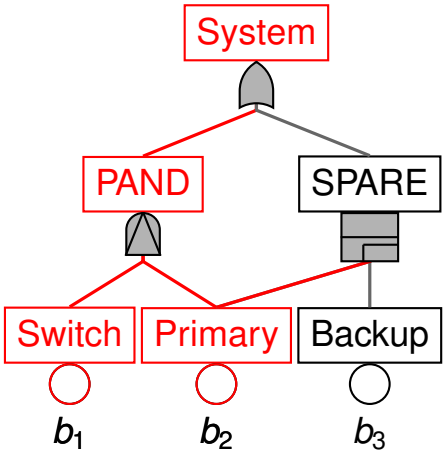
Fault Tree Examples and Challenges

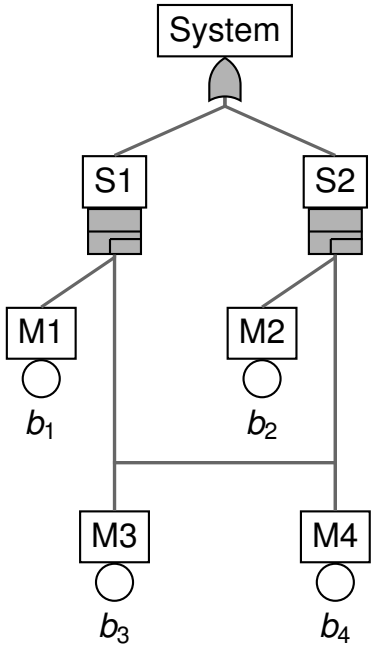


Fault Tree Examples and Challenges



Fault Tree Examples and Challenges





Problems with default semantics

- Not adaptive
- Activation order of spares may not be optimal
- Semantic issues with concurrent spare claims

Content

Non-Deterministic Fault Trees

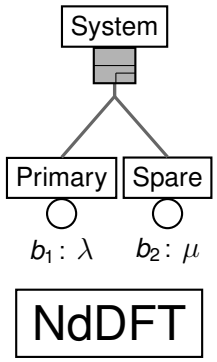
Separation of concerns

Approach: split up fault behaviour into

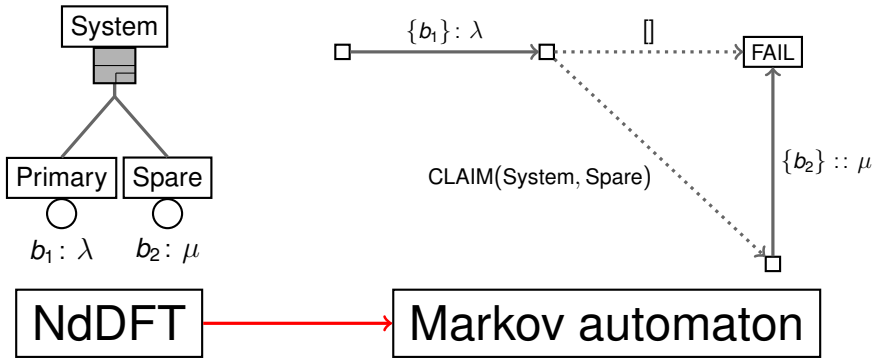
- non-deterministic fault tree
- deterministic recovery strategy

and compute recovery strategy from fault tree via optimal scheduling

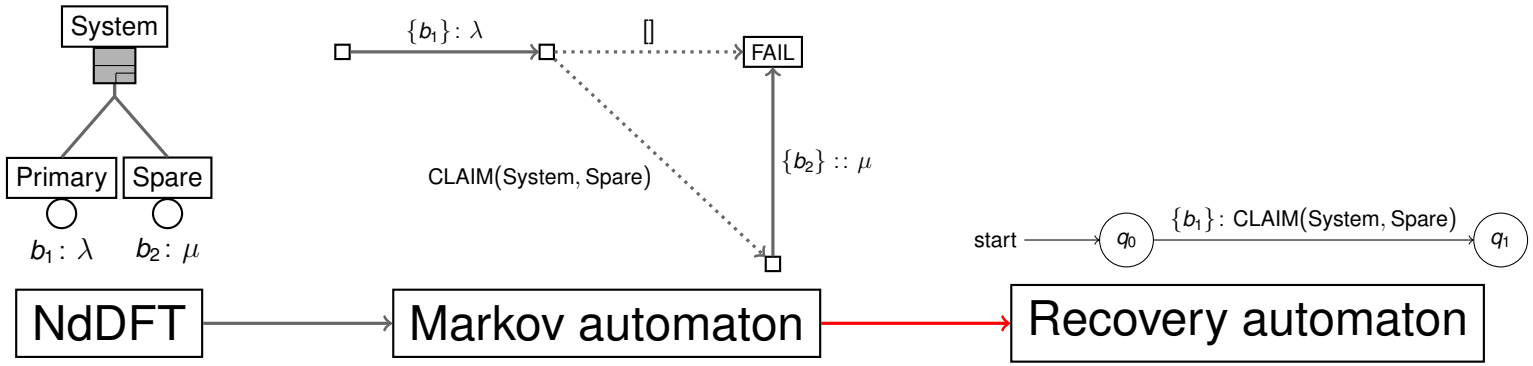
Transformation Road Map



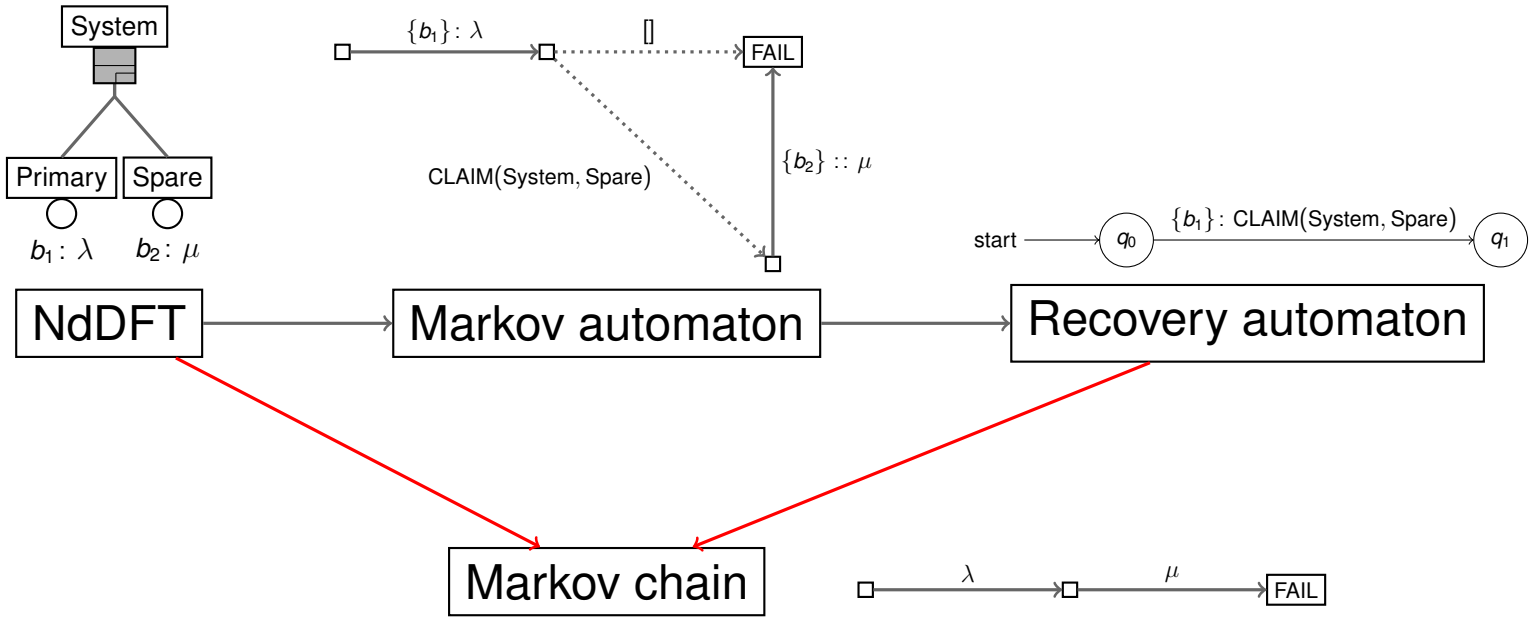
Transformation Road Map



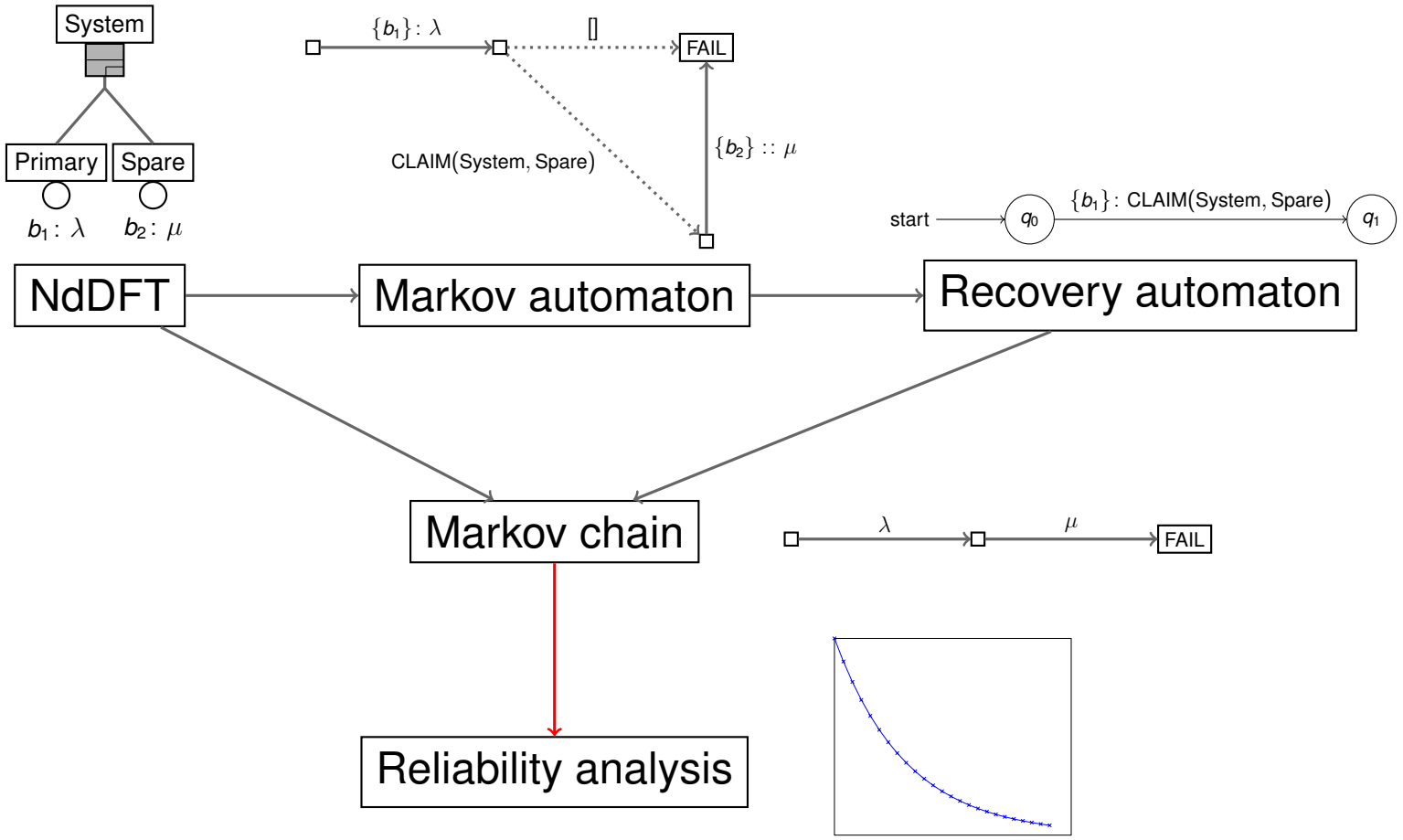
Transformation Road Map



Transformation Road Map



Transformation Road Map

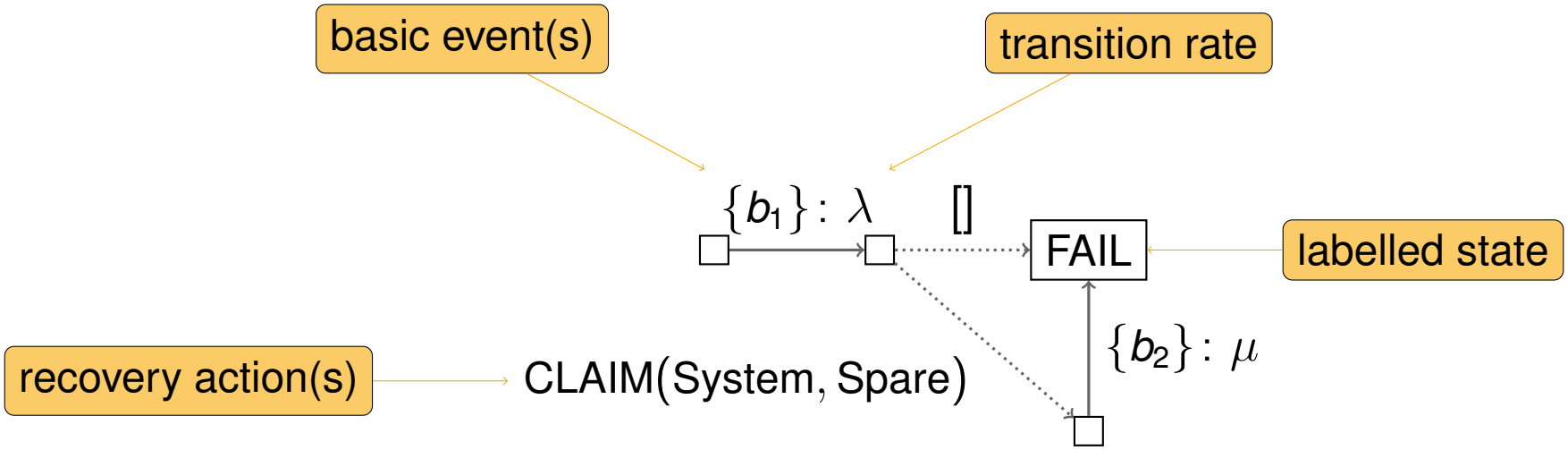


Markov Automata

Markov automaton

A Markov automaton is a finite transition system with two types of transitions:

- probabilistic, continuous-time transitions (modelling fault occurrences)
- non-deterministic, instantaneous transitions (modelling recovery actions)



Getting the Markov Automaton

Transformation of NdDFT into Markov automaton (based on [Dugan/Bavuso/Boyd 1992])

- $\mathit{States} = \mathit{EventSets}^* \times (\mathit{SpareGates} \to \mathit{SpareComponents}) \cup \{\text{FAIL}\}$
- Start in nominal state $(\varepsilon, \varepsilon)$

Getting the Markov Automaton

Transformation of NdDFT into Markov automaton (based on [Dugan/Bavuso/Boyd 1992])

- $\mathit{States} = \mathit{EventSets}^* \times (\mathit{SpareGates} \to \mathit{SpareComponents}) \cup \{\text{FAIL}\}$
- Start in nominal state $(\varepsilon, \varepsilon)$
- Repeat for any state q until full state space is generated:

Getting the Markov Automaton

Transformation of NdDFT into Markov automaton (based on [Dugan/Bavuso/Boyd 1992])

- $\mathit{States} = \mathit{EventSets}^* \times (\mathit{SpareGates} \to \mathit{SpareComponents}) \cup \{\text{FAIL}\}$
- Start in nominal state $(\varepsilon, \varepsilon)$
- Repeat for any state q until full state space is generated:
  - For each enabled basic event b, generate a q-successor q' with transition label $B : \lambda$ (where B = FDEP closure of b and λ = failure rate of b)

Getting the Markov Automaton

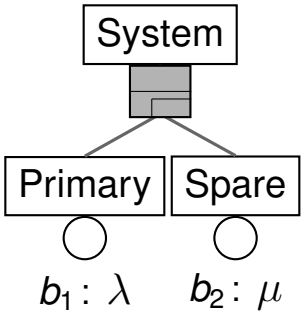
Transformation of NdDFT into Markov automaton (based on [Dugan/Bavuso/Boyd 1992])

- $\mathit{States} = \mathit{EventSets}^* \times (\mathit{SpareGates} \to \mathit{SpareComponents}) \cup \{\text{FAIL}\}$
- Start in nominal state $(\varepsilon, \varepsilon)$
- Repeat for any state q until full state space is generated:
  - For each enabled basic event b, generate a q-successor q' with transition label $B : \lambda$ (where B = FDEP closure of b and λ = failure rate of b)
    - For each enabled recovery action r, generate a q'-successor with transition label R

Getting the Markov Automaton

Transformation of NdDFT into Markov automaton (based on [Dugan/Bavuso/Boyd 1992])

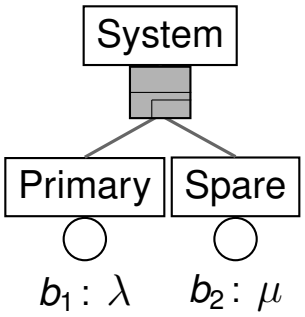
- $States = EventSets^* \times (SpareGates \to SpareComponents) \cup \{\text{FAIL}\}$
- Start in nominal state $(\varepsilon, \varepsilon)$
- Repeat for any state q until full state space is generated:
  - For each enabled basic event b, generate a q-successor q' with transition label $B : \lambda$ (where B = FDEP closure of b and λ = failure rate of b)
    - For each enabled recovery action r, generate a q'-successor with transition label R



Getting the Markov Automaton

Transformation of NdDFT into Markov automaton (based on [Dugan/Bavuso/Boyd 1992])

- $\mathit{States} = \mathit{EventSets}^* \times (\mathit{SpareGates} \to \mathit{SpareComponents}) \cup \{\text{FAIL}\}$
- Start in nominal state $(\varepsilon, \varepsilon)$
- Repeat for any state q until full state space is generated:
  - For each enabled basic event b, generate a q-successor q' with transition label $B : \lambda$ (where B = FDEP closure of b and λ = failure rate of b)
    - For each enabled recovery action r, generate a q'-successor with transition label R

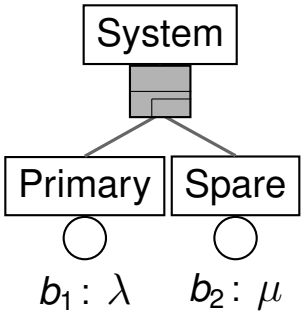


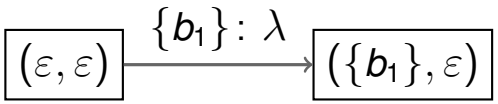
$(\varepsilon, \varepsilon)$

Getting the Markov Automaton

Transformation of NdDFT into Markov automaton (based on [Dugan/Bavuso/Boyd 1992])

- $States = EventSets^* \times (SpareGates \to SpareComponents) \cup \{\text{FAIL}\}$
- Start in nominal state $(\varepsilon, \varepsilon)$
- Repeat for any state q until full state space is generated:
  - For each enabled basic event b, generate a q-successor q' with transition label $B : \lambda$ (where B = FDEP closure of b and λ = failure rate of b)
    - For each enabled recovery action r, generate a q'-successor with transition label R

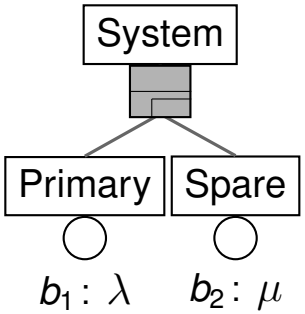


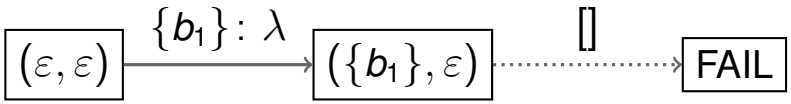


Getting the Markov Automaton

Transformation of NdDFT into Markov automaton (based on [Dugan/Bavuso/Boyd 1992])

- $States = EventSets^* \times (SpareGates \to SpareComponents) \cup \{\text{FAIL}\}$
- Start in nominal state $(\varepsilon, \varepsilon)$
- Repeat for any state q until full state space is generated:
  - For each enabled basic event b, generate a q-successor q' with transition label $B : \lambda$ (where B = FDEP closure of b and λ = failure rate of b)
    - For each enabled recovery action r, generate a q'-successor with transition label R

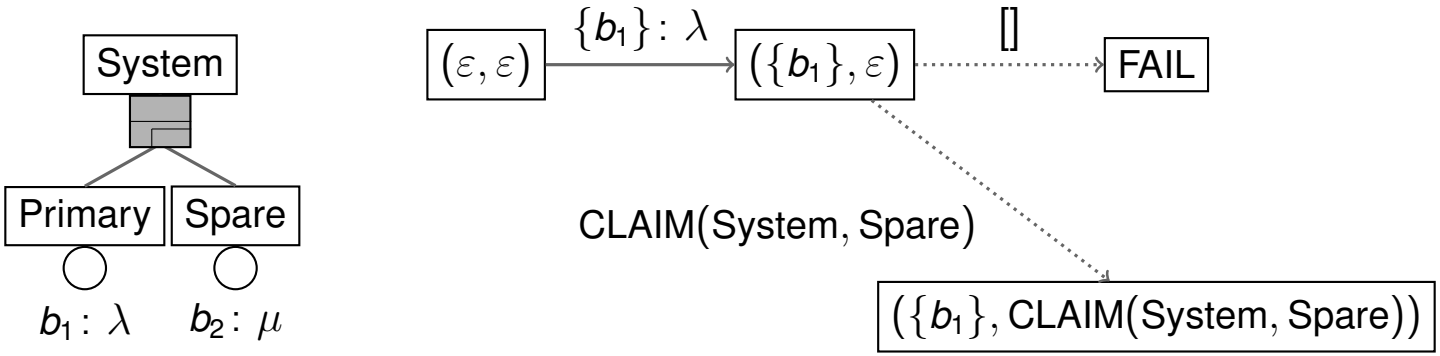




Getting the Markov Automaton

Transformation of NdDFT into Markov automaton (based on [Dugan/Bavuso/Boyd 1992])

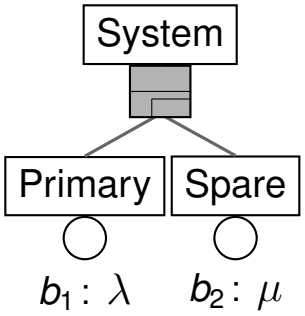
- $States = EventSets^* \times (SpareGates \to SpareComponents) \cup \{\text{FAIL}\}$
- Start in nominal state $(\varepsilon, \varepsilon)$
- Repeat for any state q until full state space is generated:
  - For each enabled basic event b, generate a q-successor q' with transition label $B : \lambda$ (where B = FDEP closure of b and λ = failure rate of b)
    - For each enabled recovery action r, generate a q'-successor with transition label R

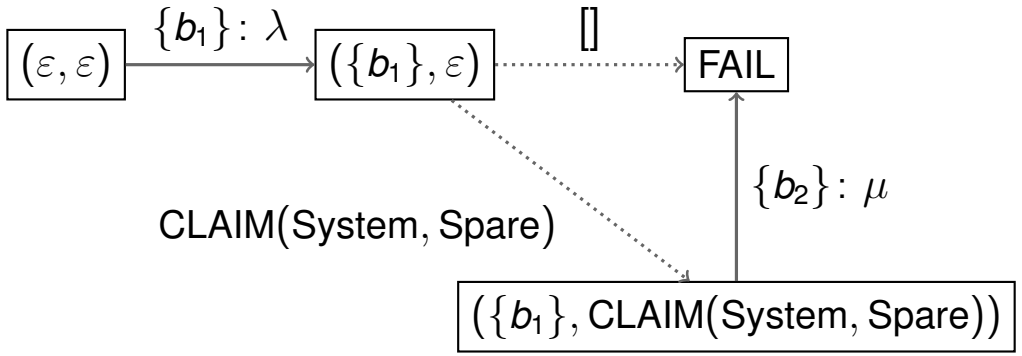


Getting the Markov Automaton

Transformation of NdDFT into Markov automaton (based on [Dugan/Bavuso/Boyd 1992])

- $States = EventSets^* \times (SpareGates \to SpareComponents) \cup \{\text{FAIL}\}$
- Start in nominal state $(\varepsilon, \varepsilon)$
- Repeat for any state q until full state space is generated:
  - For each enabled basic event b, generate a q-successor q' with transition label $B : \lambda$ (where B = FDEP closure of b and λ = failure rate of b)
    - For each enabled recovery action r, generate a q'-successor with transition label R

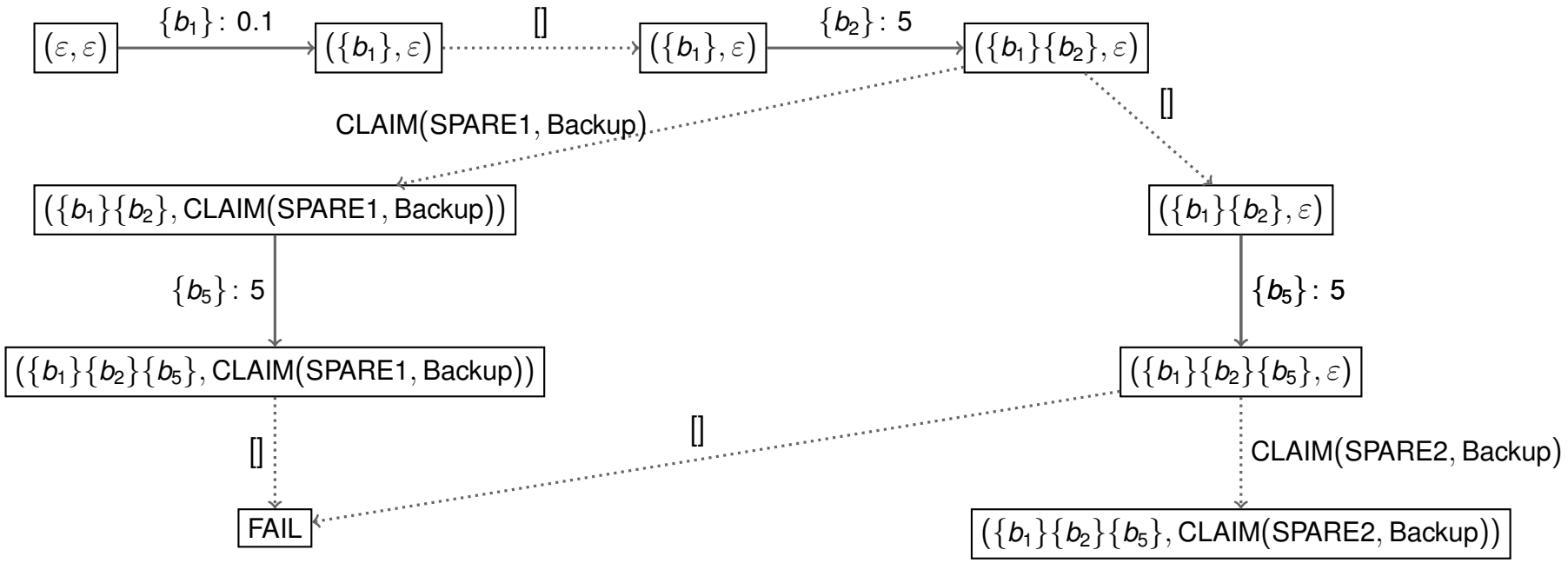




Extracting the Recovery Automaton

From Markov to recovery automaton

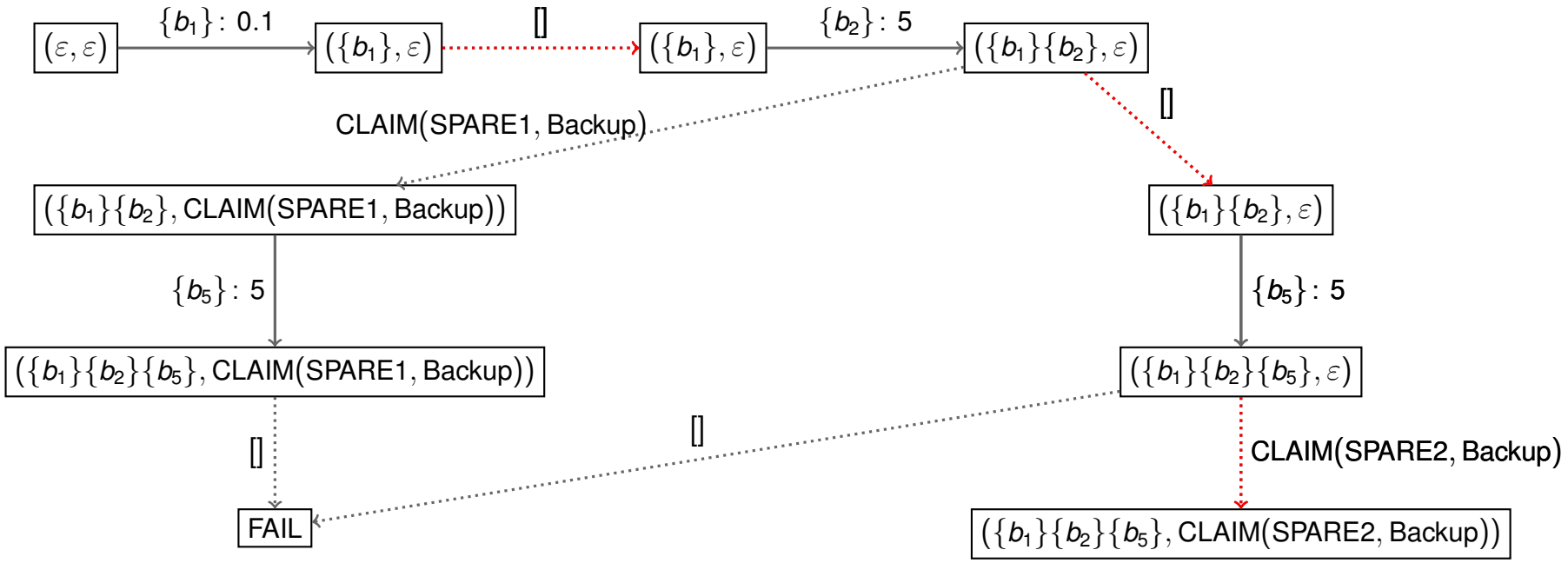
1. Start from Markov automaton



Extracting the Recovery Automaton

From Markov to recovery automaton

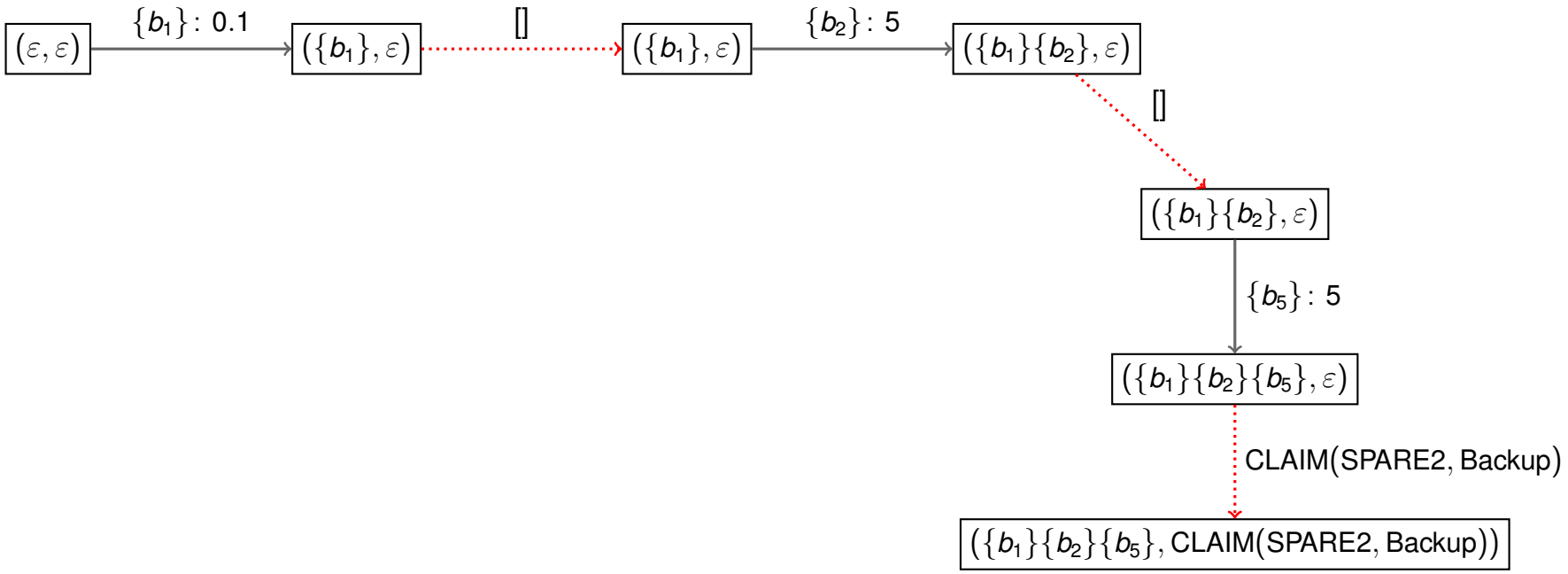
1. Start from Markov automaton
2. Compute optimal schedule (min. MTTF)



Extracting the Recovery Automaton

From Markov to recovery automaton

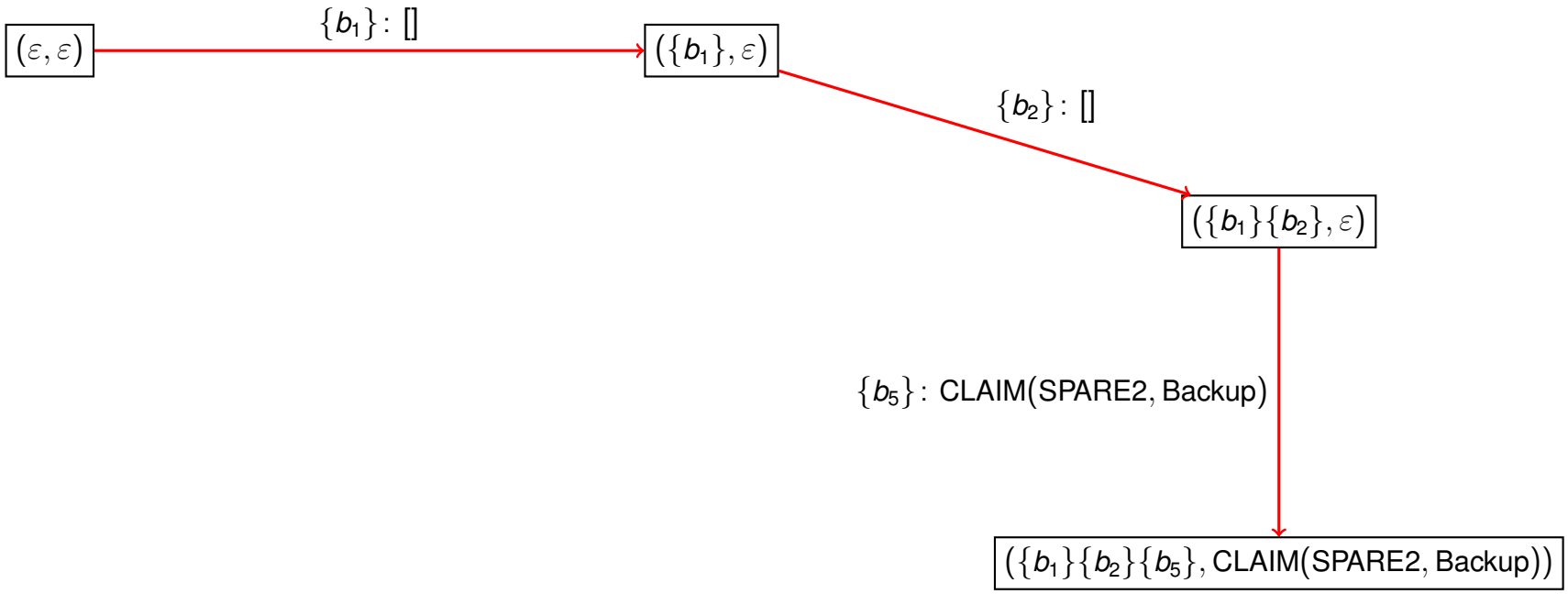
1. Start from Markov automaton
2. Compute optimal schedule (min. MTTF)
3. Remove non-scheduled transitions



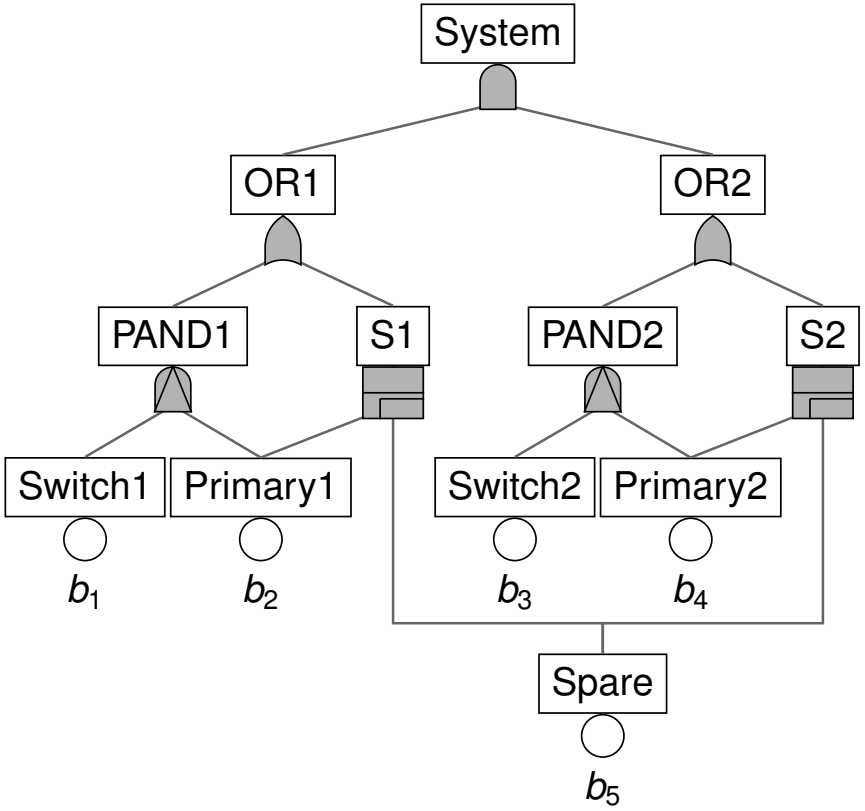
Extracting the Recovery Automaton

From Markov to recovery automaton

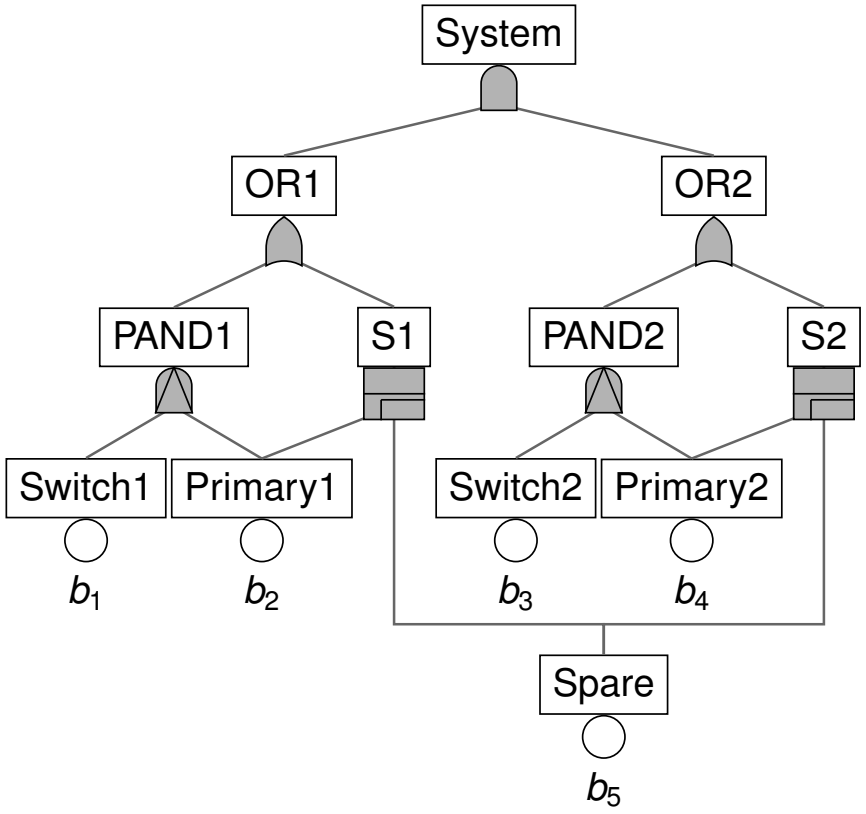
1. Start from Markov automaton
2. Compute optimal schedule (min. MTTF)
3. Remove non-scheduled transitions
4. Combine basic events and recovery actions



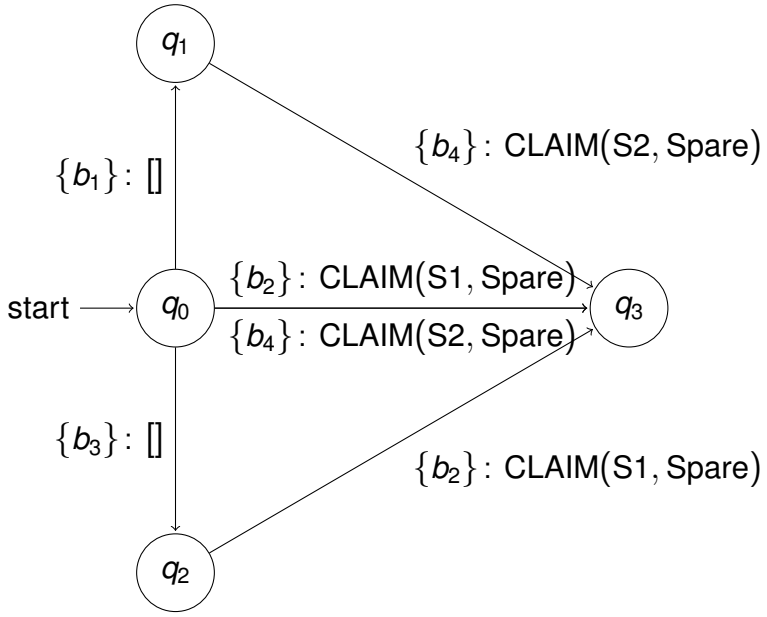
Example: Double-Redundant System



Example: Double-Redundant System



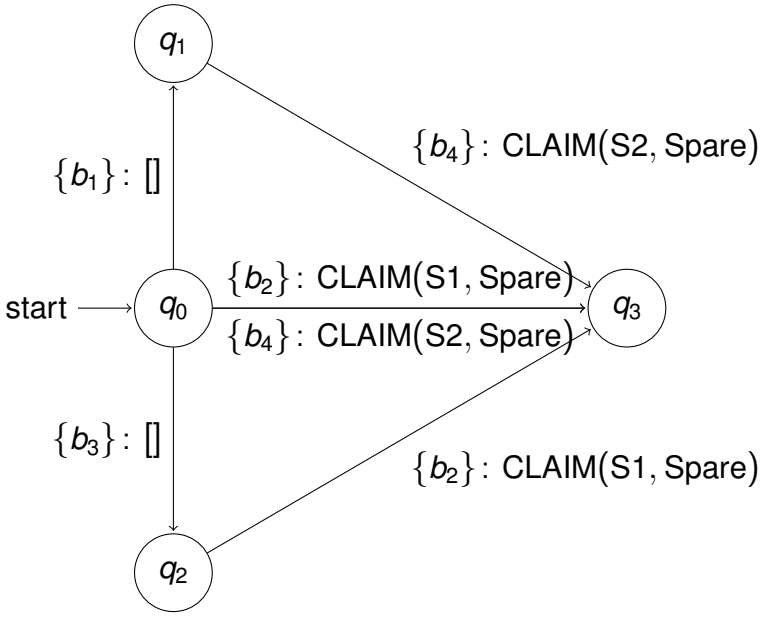
Recovery automaton (excerpt):

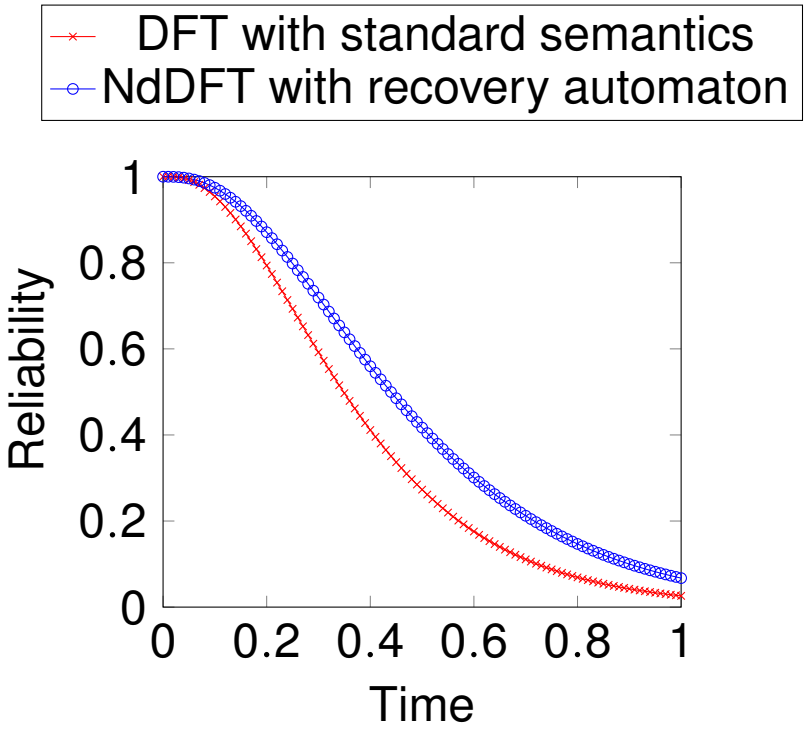


Thus: "Only claim spare if primary (and not switch) fails first"

Example: Double-Redundant System

Recovery automaton (excerpt):





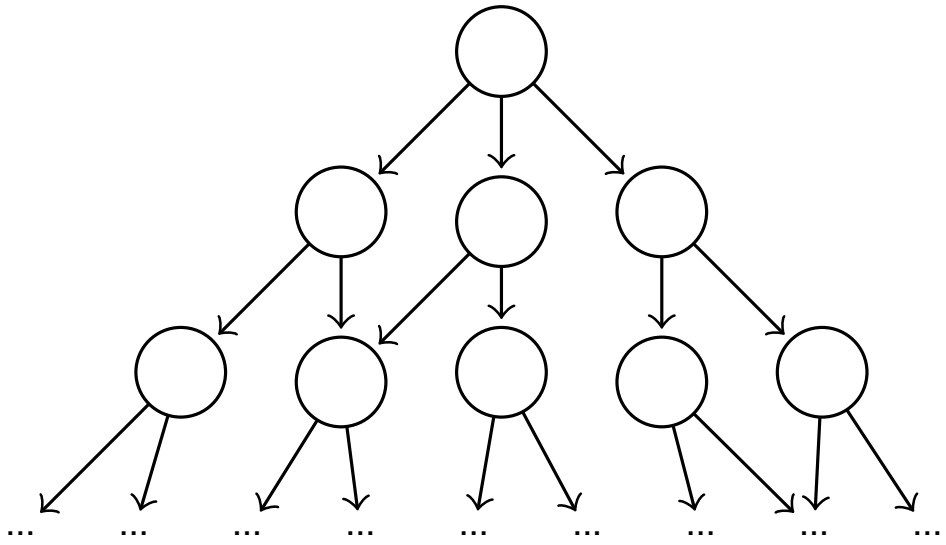
Thus: "Only claim spare if primary (and not switch) fails first"

Content

The State-Space Explosion Problem

Sources of exponential blowup

- Basic events can occur an at any time in any order! (DFT)
- Every possible recovery action causes branching! (NdDFT)



BOOOOM

The Approach

Workflow

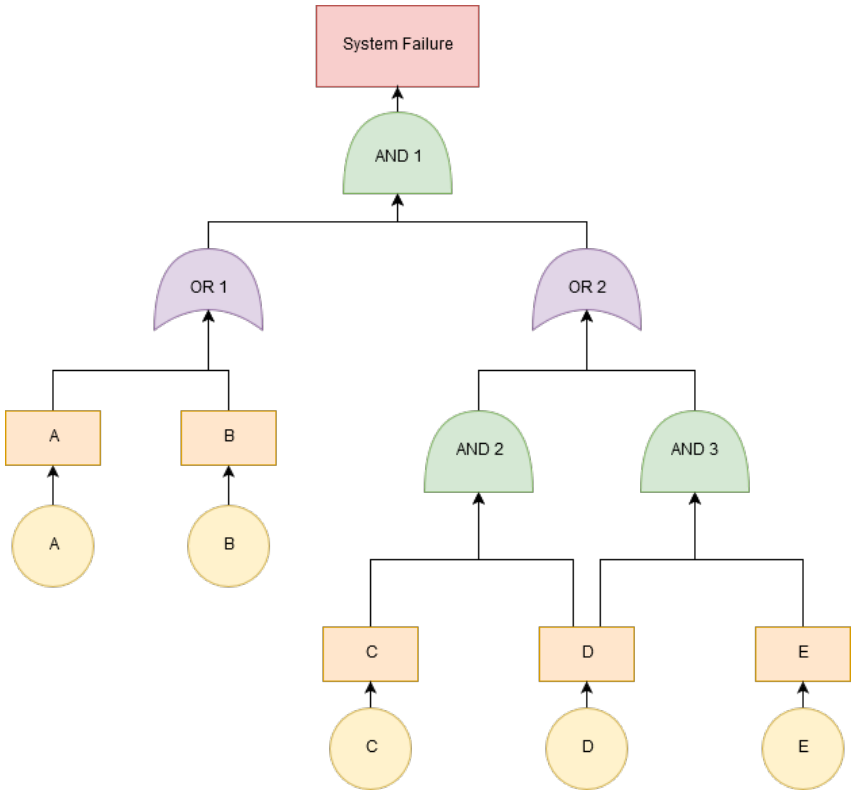
1. Modularisation of fault tree
2. Trimming of modules
3. Individual synthesis of recovery automata from modules (as seen)
4. State-space reduction of individual recovery automata
5. Composition to overall recovery automaton (parallel product)

Content

Divide and Conquer

Modules

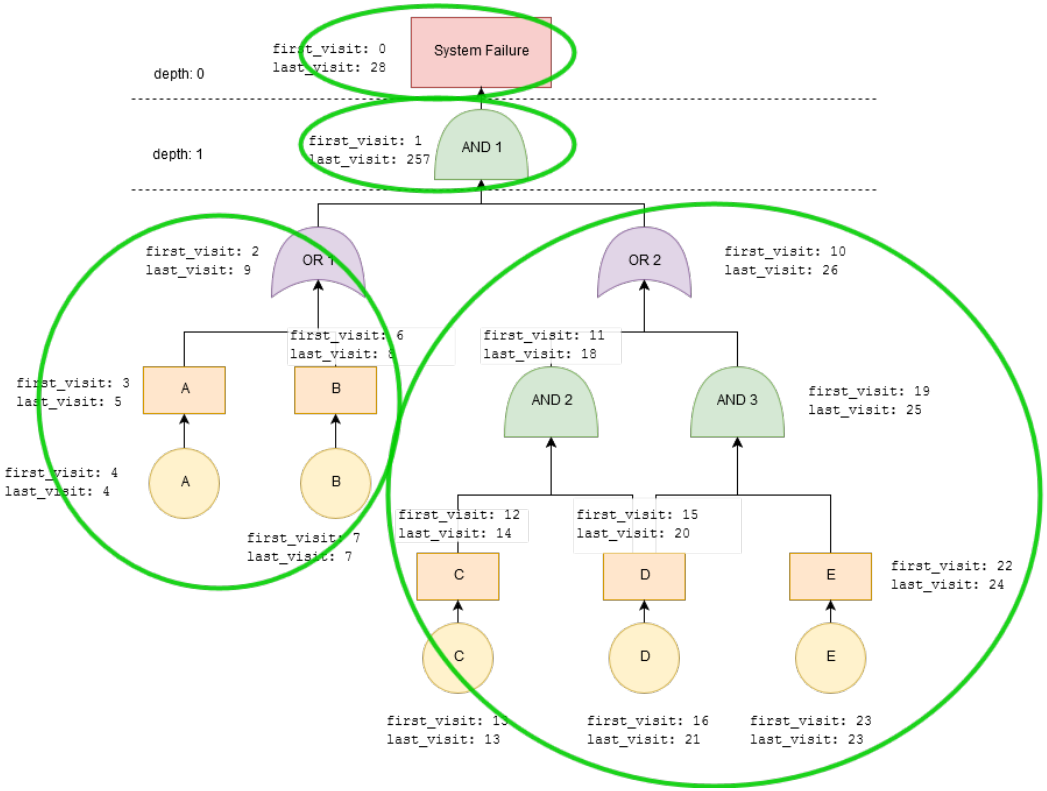
- Module = (non-trivial) subtree of fault tree that does not share basic events with other subtrees
- Thus: at least one module, identified by top-level event
- [Dutuit/Rauzy 1996] introduces linear-time algorithm for detecting modules
  - based on Tarjan algorithm for identifying SCCs



Divide and Conquer

Modules

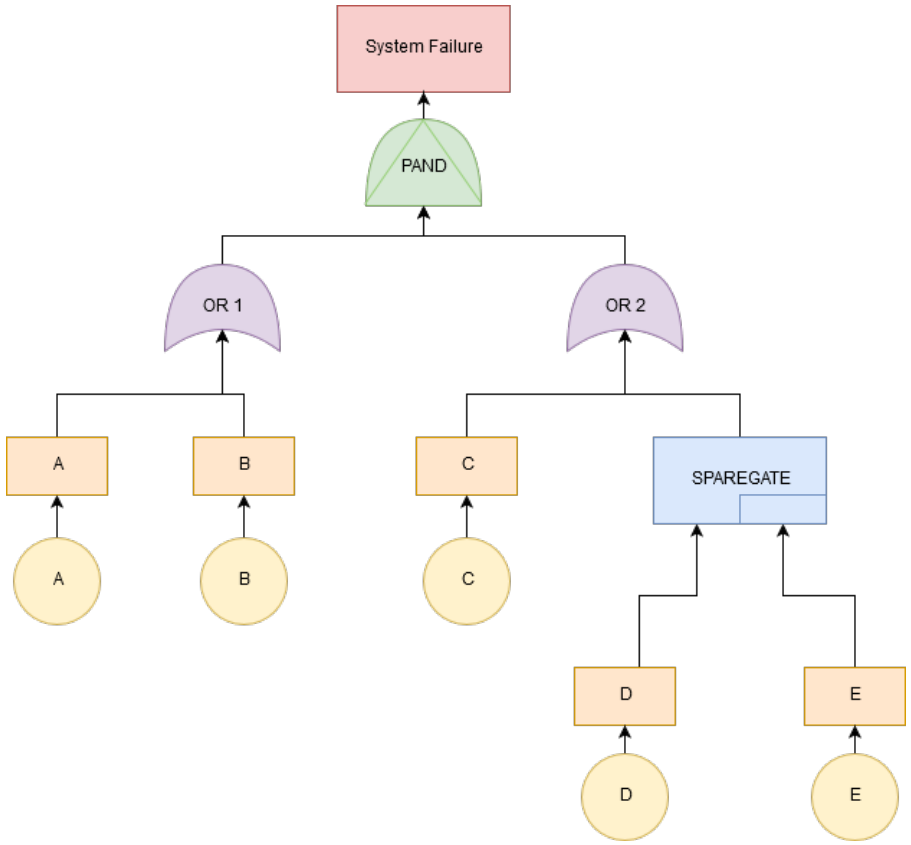
- Module = (non-trivial) subtree of fault tree that does not share basic events with other subtrees
- Thus: at least one module, identified by top-level event
- [Dutuit/Rauzy 1996] introduces linear-time algorithm for detecting modules
  - based on Tarjan algorithm for identifying SCCs



Caveat: Priority Gates

Observation

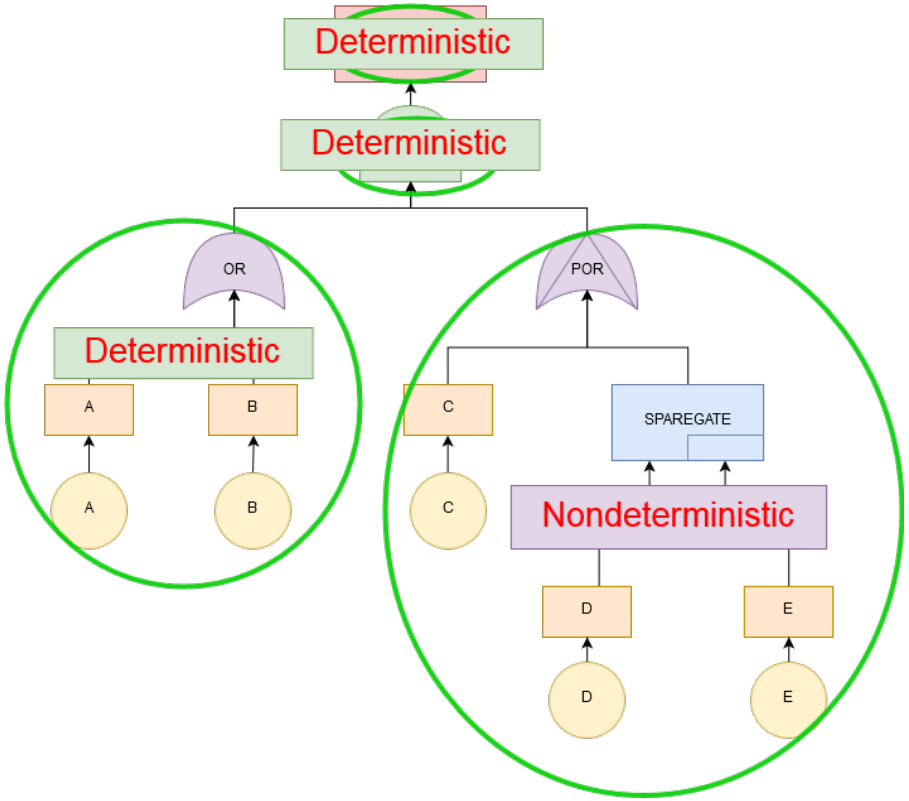
- Priority gates introduce non-monotonic behaviour: avoiding failure of subsystem can decrease overall reliability
- Therefore: exclude non-left subtrees of priority gates that contain SPARE gates



Trimming

Observation

- Modules without SPARE gates do not affect recovery
- Can be ignored in construction process (but not in subsequent reliability analysis)



FFORT Benchmark Suite

FFORT

FFORT (*the Fault tree FOResT, formerly the Twente Arberretum*) is a collection of fault trees, collected from scientific literature.

Our main purpose is to provide a benchmark suite, so that researchers on fault tree analysis can use a large and diverse number of fault trees to test and validate their methods and tools. Each fault tree is given in an extended Galileo format (described here), and results for earlier analyses are provided, if available.

Clicking on the fault tree's name, will provide additional information, including the paper where the model first appeared, and a graphical representation.

Statistics about the fault trees collected in FFORT can be found here.

All trees (including metadata) are available in our git repository. To obtain this repository, enter *git clone https://dftbenchmarks.utwente.nl/ffort.git*

To submit a new model, please go here to provide the FT in Galileo format, and its associated metadata, and send it to the maintainers.

Search

Show fault trees with name... of type: Any with description containing...

With at least one of these gate types: ☐AND ☐FDEP ☐OR ☐SEQ ☐VOT ☐PAND ☐SPARE ☐IM

With at least one of these result types: ☐MTTF ☐Unreliability ☐Unavailability

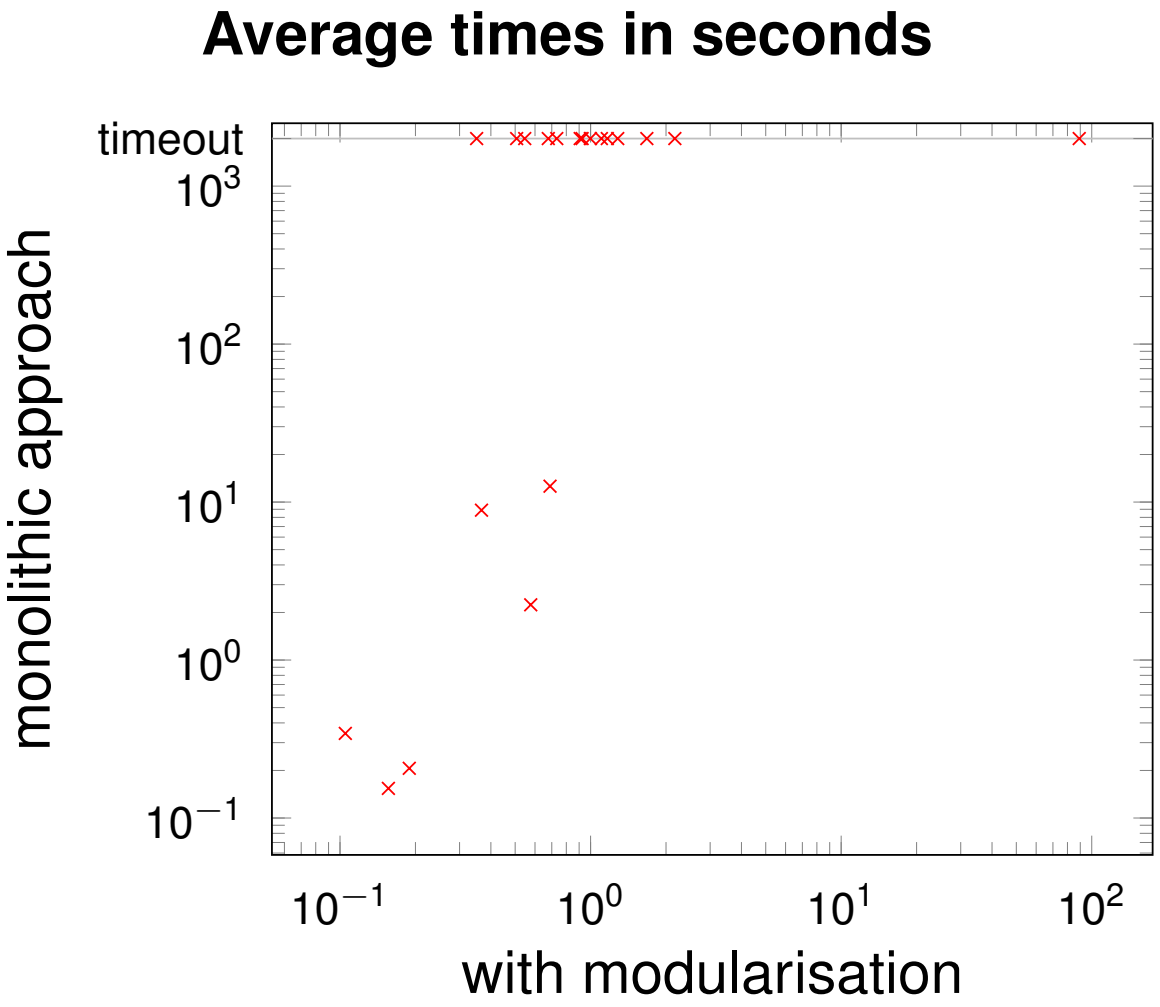
With this many basic elements: minimum ≤ |BE| ≤ maximum

With this many gates: minimum ≤ |Gates| ≤ maximum

Published by author... After year... and added to FFORT After TT . MM . JJJJ

Name▲	# BEs	# Gates	Distr.	Gate Types	Repair	Model	Results
Active Heat Rejection System (AHRS)	6–8	5–6	Exp	AND, FDEP, OR, SPARE	No	[show 2 variations]	MTTF, Unreliability
Cardiac Assist System	9–10	8–10	Exp	AND, FDEP, OR, PAND, SPARE	No	[show 2 variations]	MTTF, Unreliability
Container Seal Design Example	6	4	Discr	AND, OR	No	csd/csd	Unreliability
Dual processor reactor regulation system	40	14	Exp	AND, OR, PAND, SEQ	No	dphs/dphs_rss	MTTF, Unreliability
Electro-Mechanical Actuator	20	15	Exp	AND, FDEP, OR, SEQ, SPARE	No	ema/EMA	MTTF, Unreliability
Fault tolerant parallel processor	8–20	9–34	Exp	IM, OR, SPARE, VOT	Varies	[show 16 variations]	MTTF, Unavailability, Unreliability

Effect of Modularisation



Content

Minimising Recovery Automata

- “Standard” state-space reduction can be done with trace equivalence.
- **Goal:** exploit special properties of recovery automata (persistency of faults) to go further

Minimising Recovery Automata

- "Standard" state-space reduction can be done with trace equivalence.
- **Goal:** exploit special properties of recovery automata (persistency of faults) to go further

Definition (Automata equivalences)

Let $\mathfrak{A}, \mathfrak{B}$ be recovery automata with output functions of type

$$\rho_{\mathfrak{A}}, \rho_{\mathfrak{B}} : \mathit{EventSets}^* \to \mathit{RecoveryActions}^*.$$

- $\mathfrak{A}$ and $\mathfrak{B}$ are called trace equivalent
  if $\rho_{\mathfrak{A}}(B_1 \cdot \ldots \cdot B_n) = \rho_{\mathfrak{B}}(B_1 \cdot \ldots \cdot B_n)$ for all $B_1, \ldots, B_n$.
- $\mathfrak{A}$ and $\mathfrak{B}$ are called recovery equivalent
  if $\rho_{\mathfrak{A}}(B_1 \cdot \ldots \cdot B_n) = \rho_{\mathfrak{B}}(B_1 \cdot \ldots \cdot B_n)$ for all $B_1, \ldots, B_n$ with $B_i \cap B_j = \emptyset$ for $i \neq j$.

Minimising Recovery Automata

- "Standard" state-space reduction can be done with trace equivalence.
- **Goal:** exploit special properties of recovery automata (persistency of faults) to go further

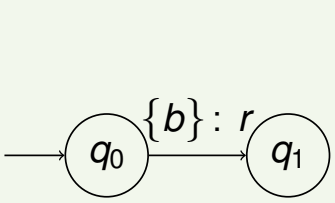
Definition (Automata equivalences)

Let $\mathfrak{A}, \mathfrak{B}$ be recovery automata with output functions of type

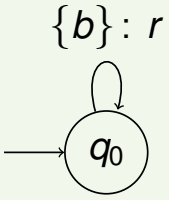
$$\rho_{\mathfrak{A}}, \rho_{\mathfrak{B}} : \mathit{EventSets}^* \to \mathit{RecoveryActions}^*.$$

- $\mathfrak{A}$ and $\mathfrak{B}$ are called trace equivalent
  if $\rho_{\mathfrak{A}}(B_1 \cdot \ldots \cdot B_n) = \rho_{\mathfrak{B}}(B_1 \cdot \ldots \cdot B_n)$ for all $B_1, \ldots, B_n$.
- $\mathfrak{A}$ and $\mathfrak{B}$ are called recovery equivalent
  if $\rho_{\mathfrak{A}}(B_1 \cdot \ldots \cdot B_n) = \rho_{\mathfrak{B}}(B_1 \cdot \ldots \cdot B_n)$ for all $B_1, \ldots, B_n$ with $B_i \cap B_j = \emptyset$ for $i \neq j$.

Example (Trace vs. recovery equivalence)



and



are recovery (but not trace) equivalent

Merging Orthogonal States

Idea: merge states for which all disagreeing output transitions are never enabled

Definition (Orthogonality)

Recovery automaton states q_1 and q_2 are orthogonal with respect to event set B if

- on every path to q_1, some event in B must occur or
- on every path to q_2, some event in B must occur

Merging Orthogonal States

Idea: merge states for which all disagreeing output transitions are never enabled

Definition (Orthogonality)

Recovery automaton states q_1 and q_2 are orthogonal with respect to event set B if

- on every path to q_1, some event in B must occur or
- on every path to q_2, some event in B must occur

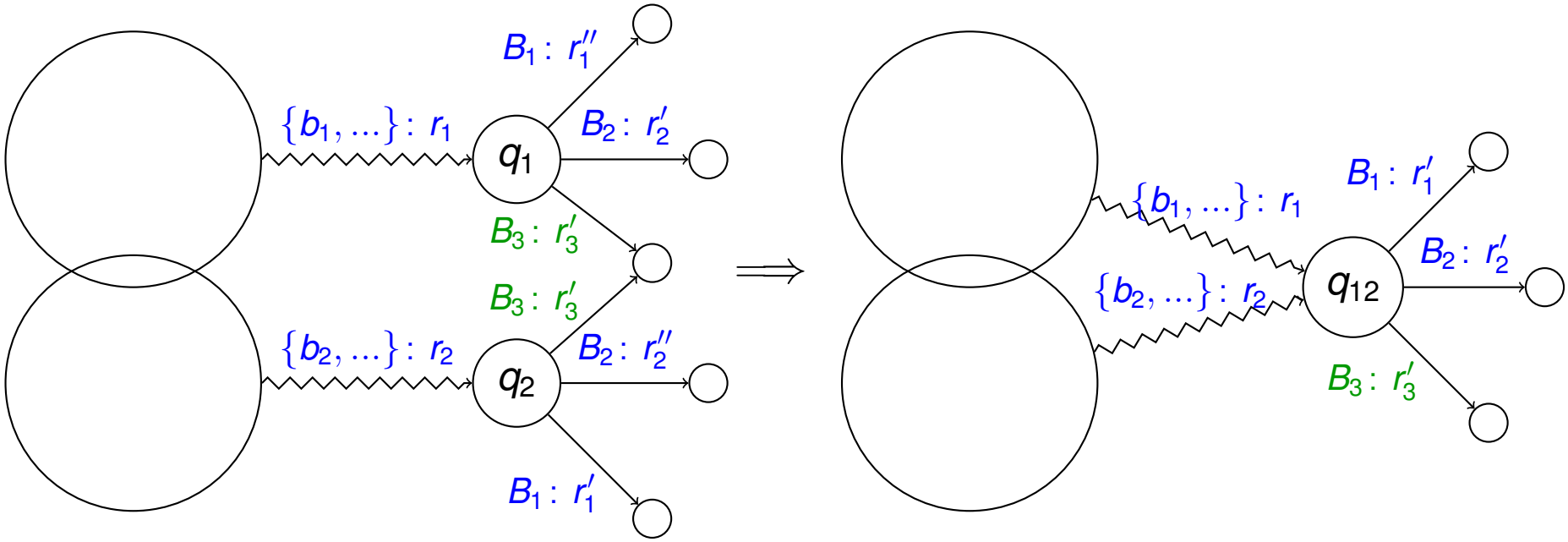
Observation: q_1 and q_2 can be merged if for all "outgoing" events sets B,

- q_1 and q_2 are orthogonal with respect to B or
- $\delta(q_1, B) = (q_1', r_1)$ and $\delta(q_2, B) = (q_2', r_2)$ such that q_1' and q_2' can be merged and $r_1 = r_2$

Merging Orthogonal States

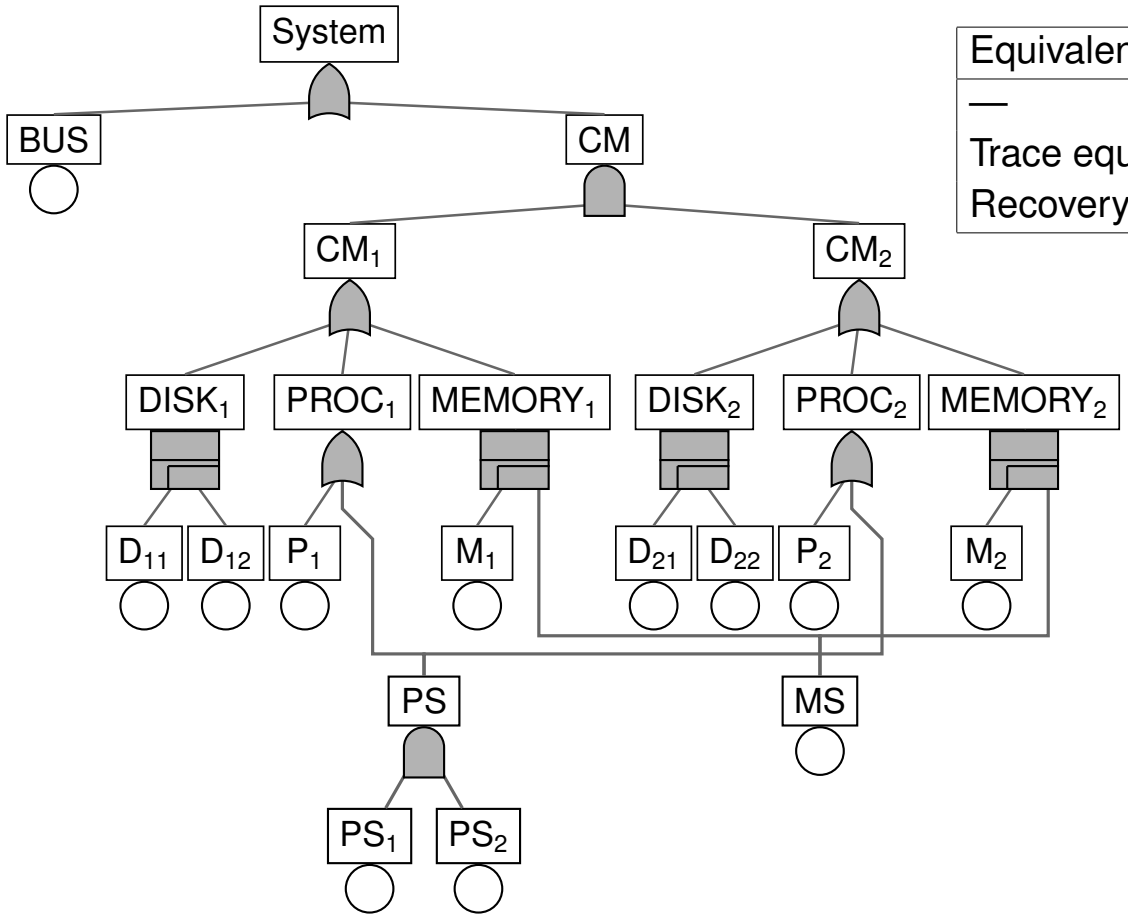
Observation: q_1 and q_2 can be merged if for all "outgoing" events sets B,

- q_1 and q_2 are orthogonal with respect to B or
- $\delta(q_1, B) = (q_1', r_1)$ and $\delta(q_2, B) = (q_2', r_2)$ such that q_1' and q_2' can be merged and $r_1 = r_2$



(where $b_1 \in B_1$ and $b_2 \in B_2$)

Evaluation on Multiprocessor Computing System



Equivalence relation	#States	States removed
—	991	—
Trace equivalence	67	93.24%
Recovery equivalence	41	95.86%

Content

Conclusion

What we have done

- Separation of concerns: split DFT into
  - NdDFT: fault effects and possible recovery actions and
  - Recovery automaton: deterministic recovery strategy
- Automated synthesis of recovery automaton
- Exploitation of fault tree properties for modularisation and state-space reduction
  - not shown: normal form and additional transformation to achieve minimality

Conclusion

What we have done

- Separation of concerns: split DFT into
  - NdDFT: fault effects and possible recovery actions and
  - Recovery automaton: deterministic recovery strategy
- Automated synthesis of recovery automaton
- Exploitation of fault tree properties for modularisation and state-space reduction
  - not shown: normal form and additional transformation to achieve minimality

Ongoing and future work

- Generalisation of recovery automata to Mealy machines with dependent inputs [Mileva 2019]
- Extension of model to include repair events and partial observability
- Corresponding adaptation of modularisation and minimisation algorithm
- Integration in VirSat FDIR (based on Storm)