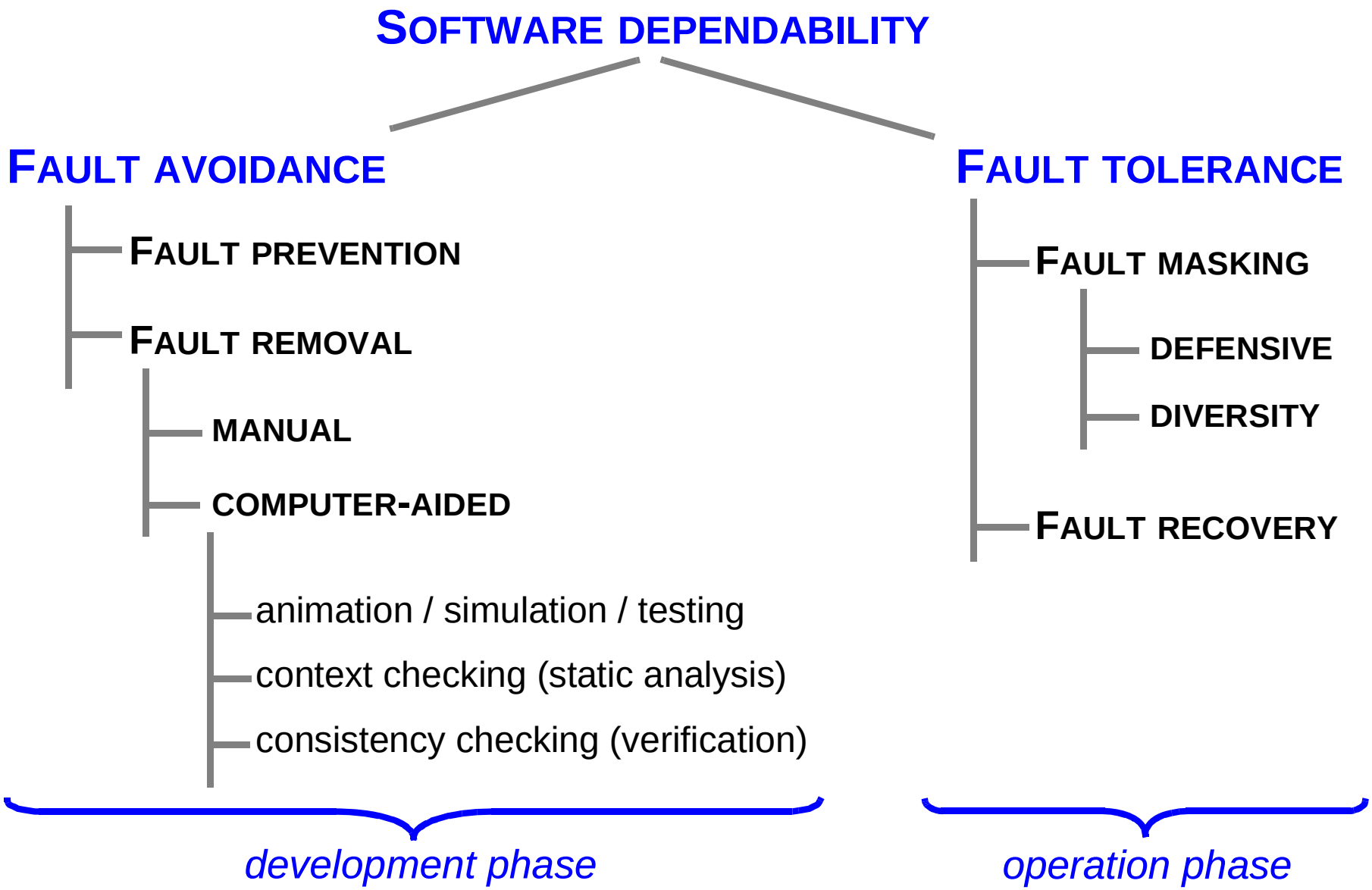
SYSTEMATIC TEST AND VERIFICATION OF SOFTWARE-BASED SYSTEMS

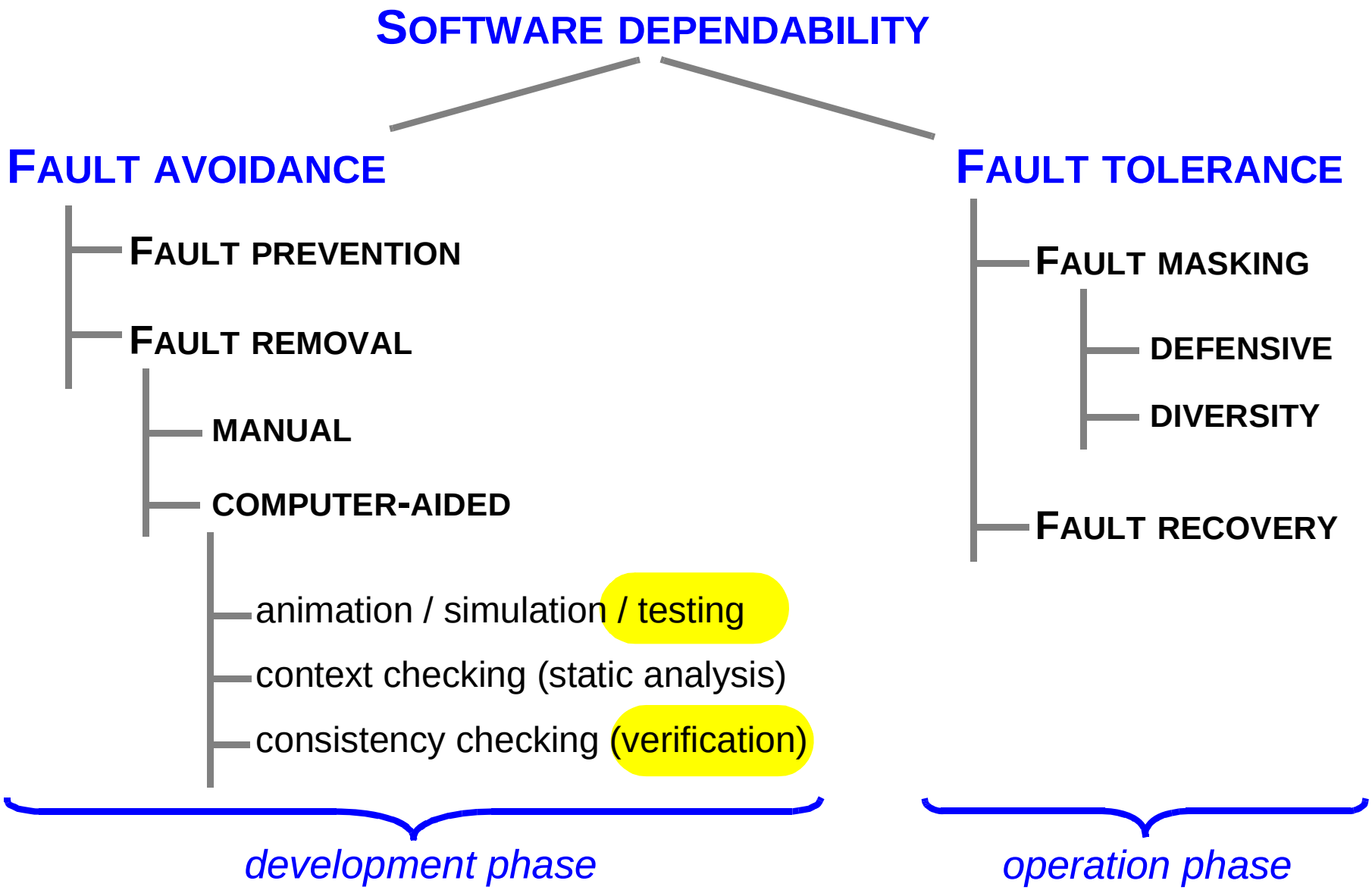
Monika Heiner

BTU Cottbus,
Dep. of Computer Science

METHODS TAXONOMY

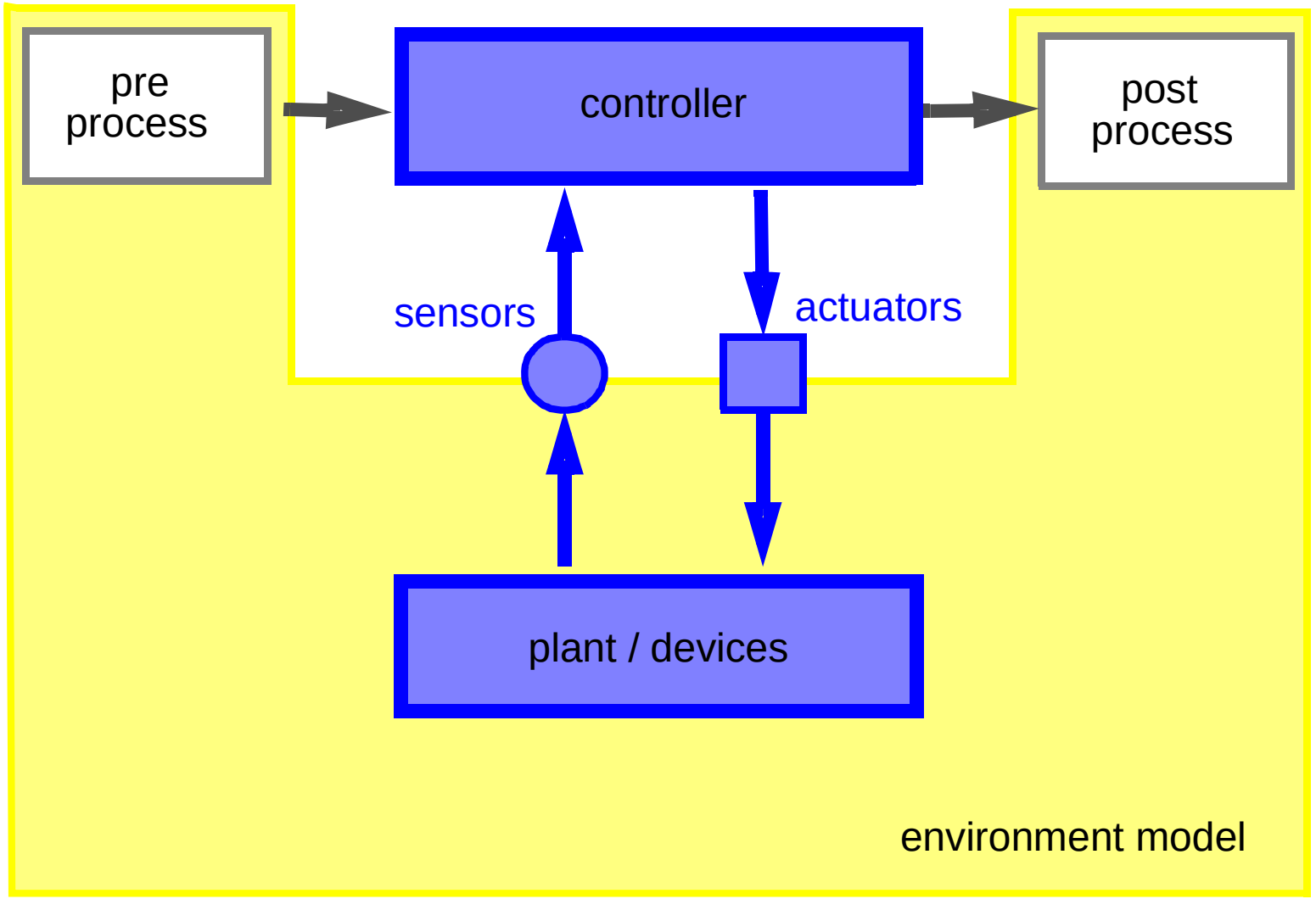


METHODS TAXONOMY

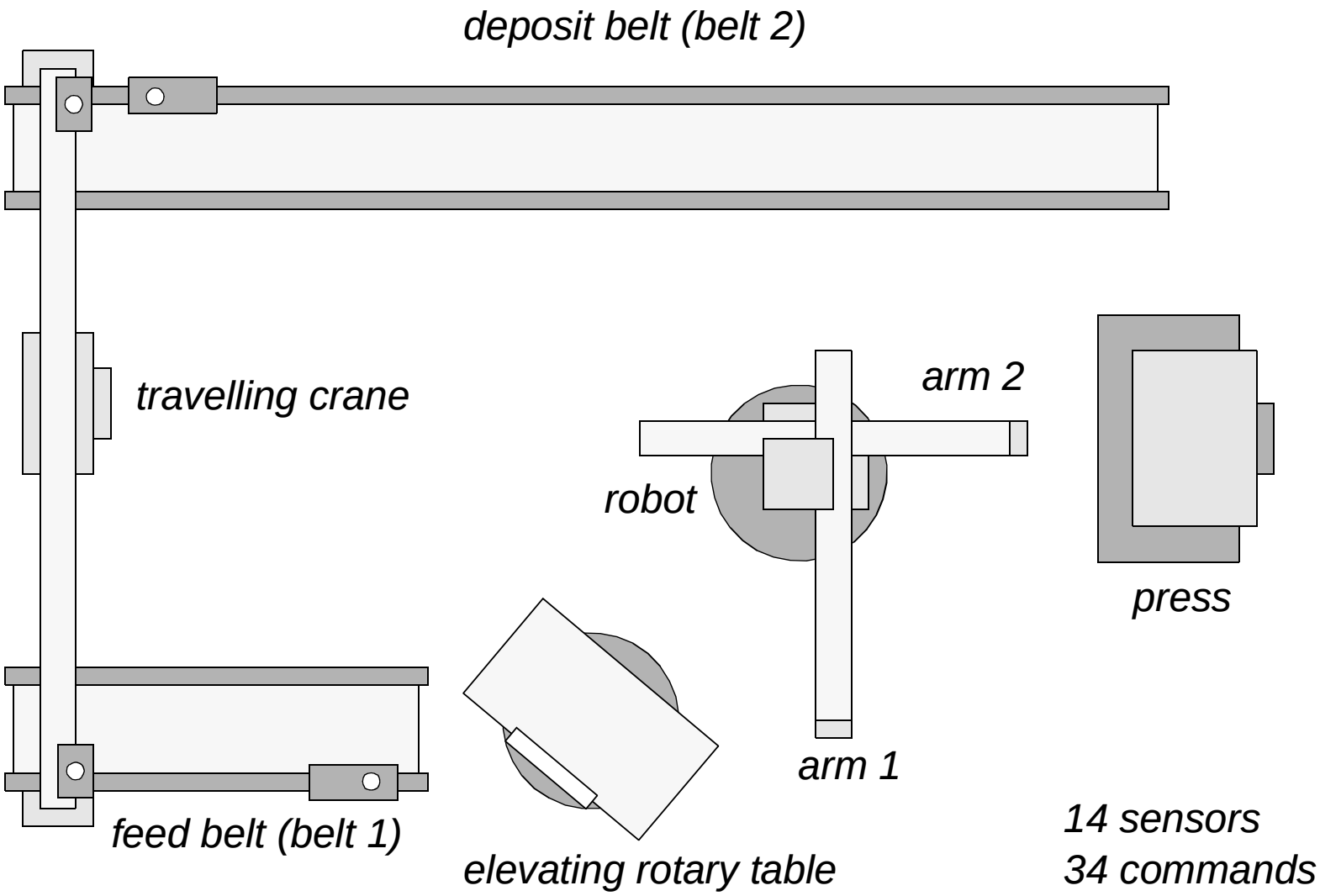


Application Scenario

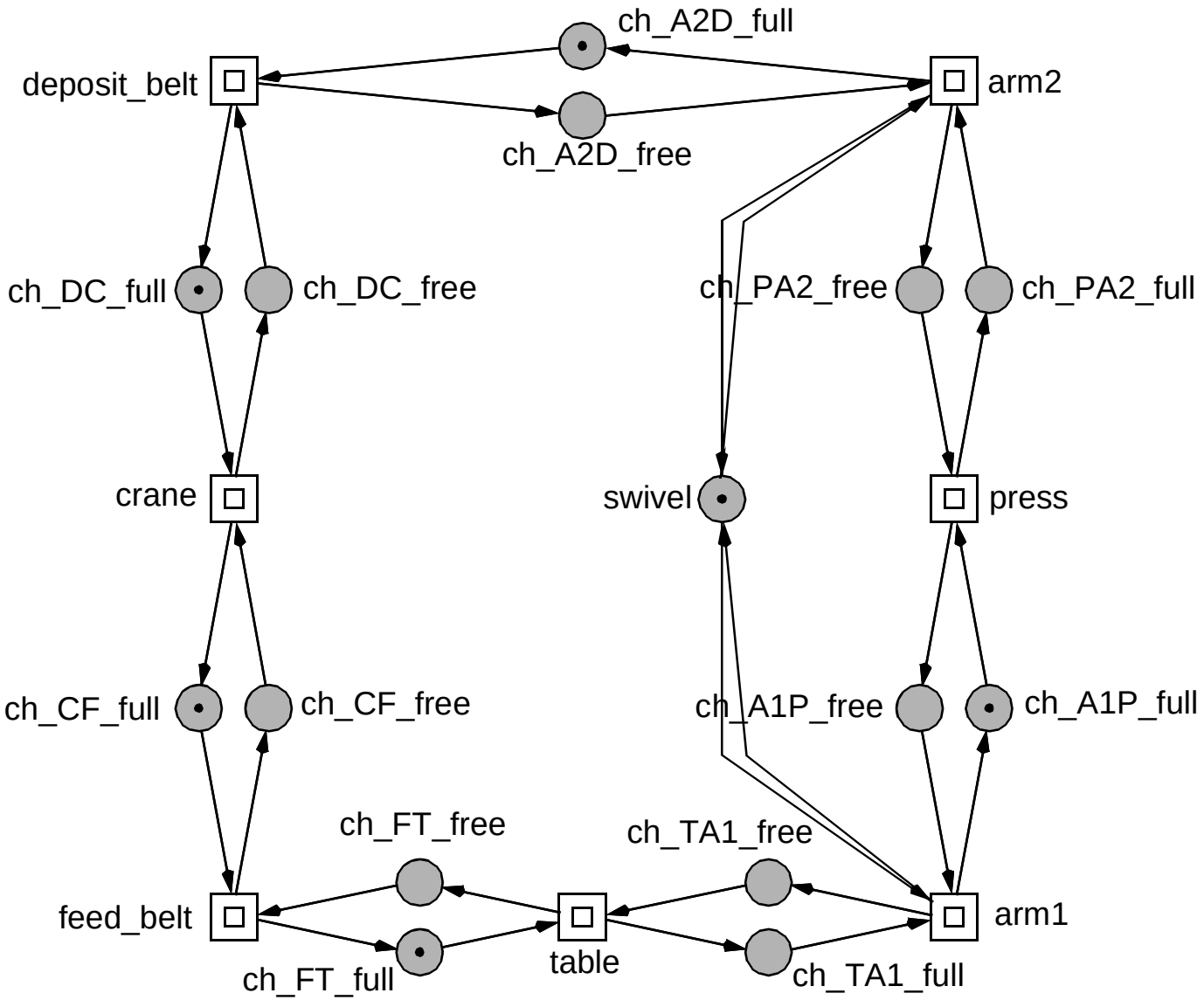
REACTIVE SYSTEMS, BASIC STRUCTURE



CASE STUDY - PRODUCTION CELL

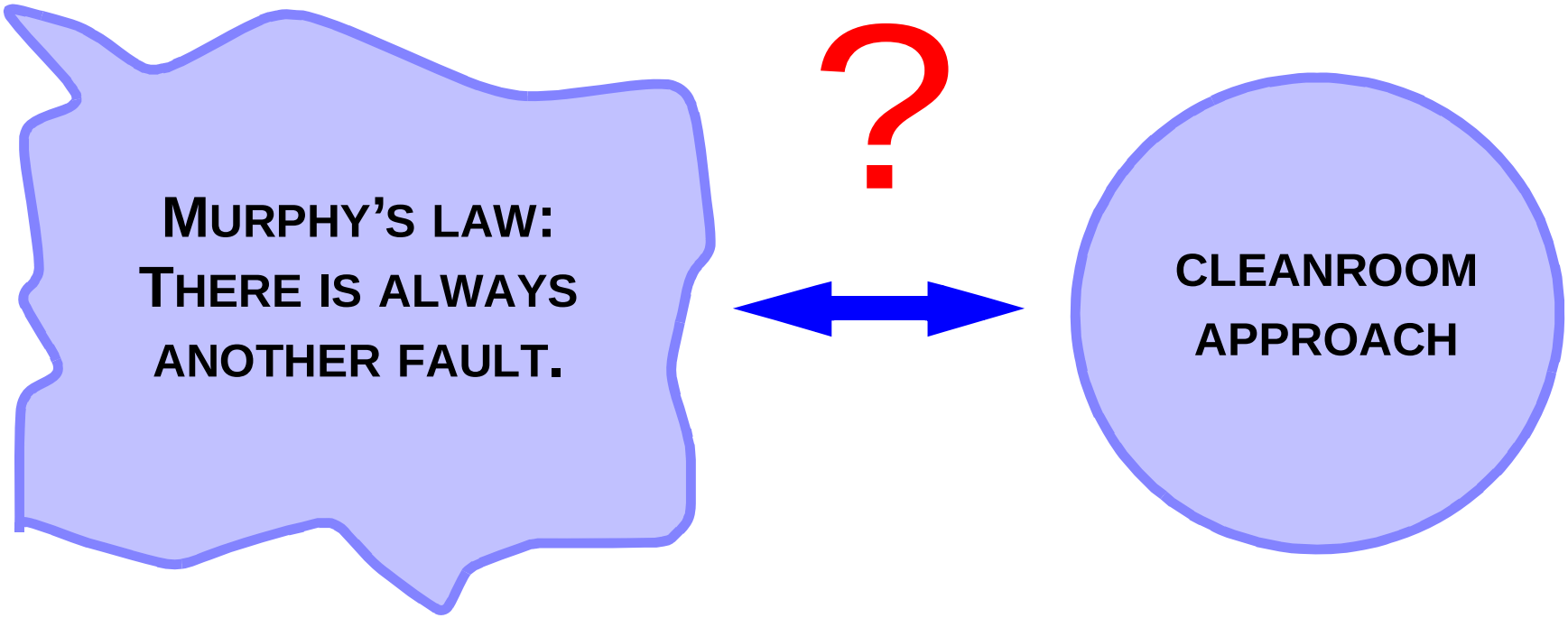


CLOSED SYSTEM, COARSE STRUCTURE



231 P,
202 T,
65 PAGES

STATE OF THE ART



STATE OF THE ART

- **natural fault rate of seasoned programmers**
  - about 1-3 % of produced program lines

- **undecidability of basic questions in software validation**
  -> *program termination*
  -> *equivalence of programs*
  -> *program verification*
  . . .

- **validation = testing**

- **testing portion of total software production effort**
  -> *standard system:* $\geq 50\ \%$
  -> *extreme availability demands:* $\approx 80\ \%$

LIMITATIONS OF TESTING

- **"Program testing can be used to show the presence of bugs, but never to show their absence !" [Dijkstra 72]**

- **exhaustive testing impossible**
  - -> *all valid inputs* -> *correctness, . . .*
  - -> *all invalid inputs* -> *robustness, security, reliability*
  - -> *state-preserving software (OS/IS):*
    a (trans-) action depends on its predecessors -> *all possible state sequences*

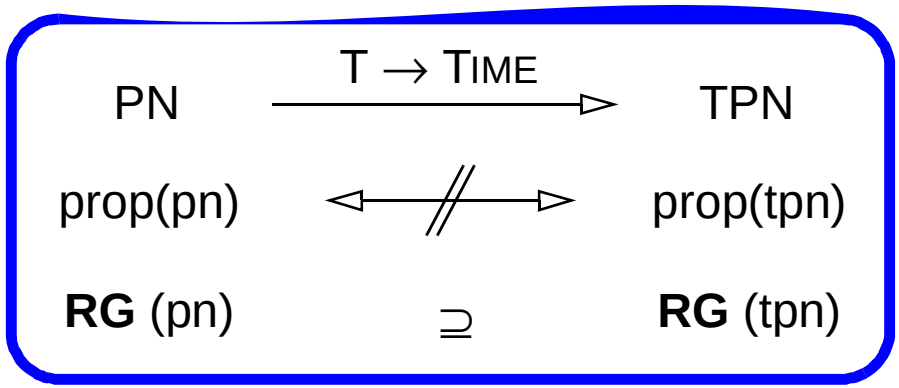
- **"Testing means the execution of a program in order to find bugs." [Myers 79]**
  - -> *A test run is called successful,*
    if it discovers unknown bugs, else unsuccessful.

- **testing is an inherently destructive task**
  - -> *most programmers unable to test own programs*

- systematic testing of concurrent programs
  is much more complicated than of sequential ones

- **state space explosion,**
  worst-case: product of the sequential state spaces

- **probe effect**
  - -> *system exhibits in test mode other (less) behavior than in standard mode*
  - -> *test means (debugger) affect timing behavior*



result:

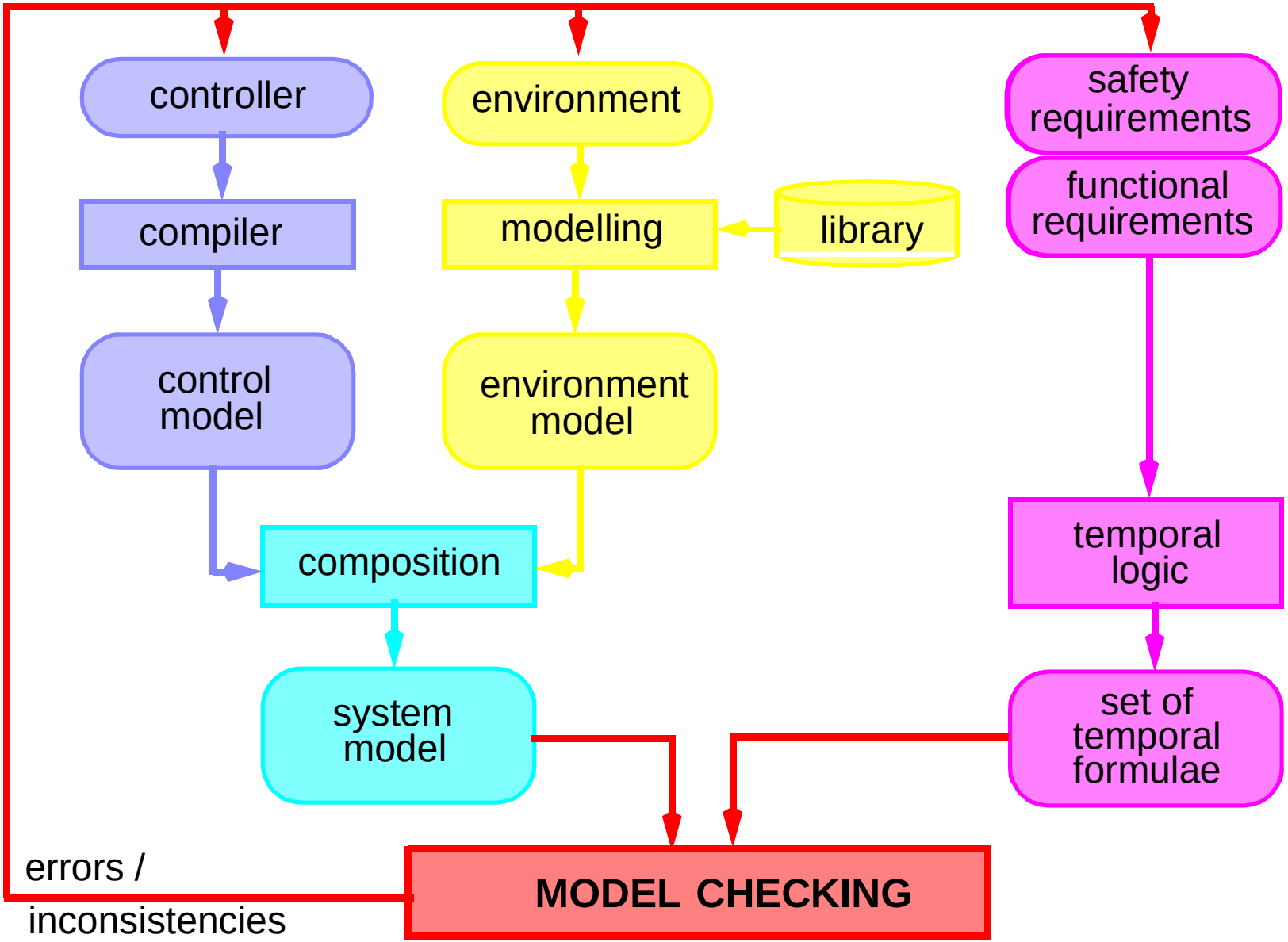
masking of certain types of bugs:

DSt (pn)-> not DSt (tpn)
live(pn) -> not live (tpn)
not BND (pn) -> BND (tpn)

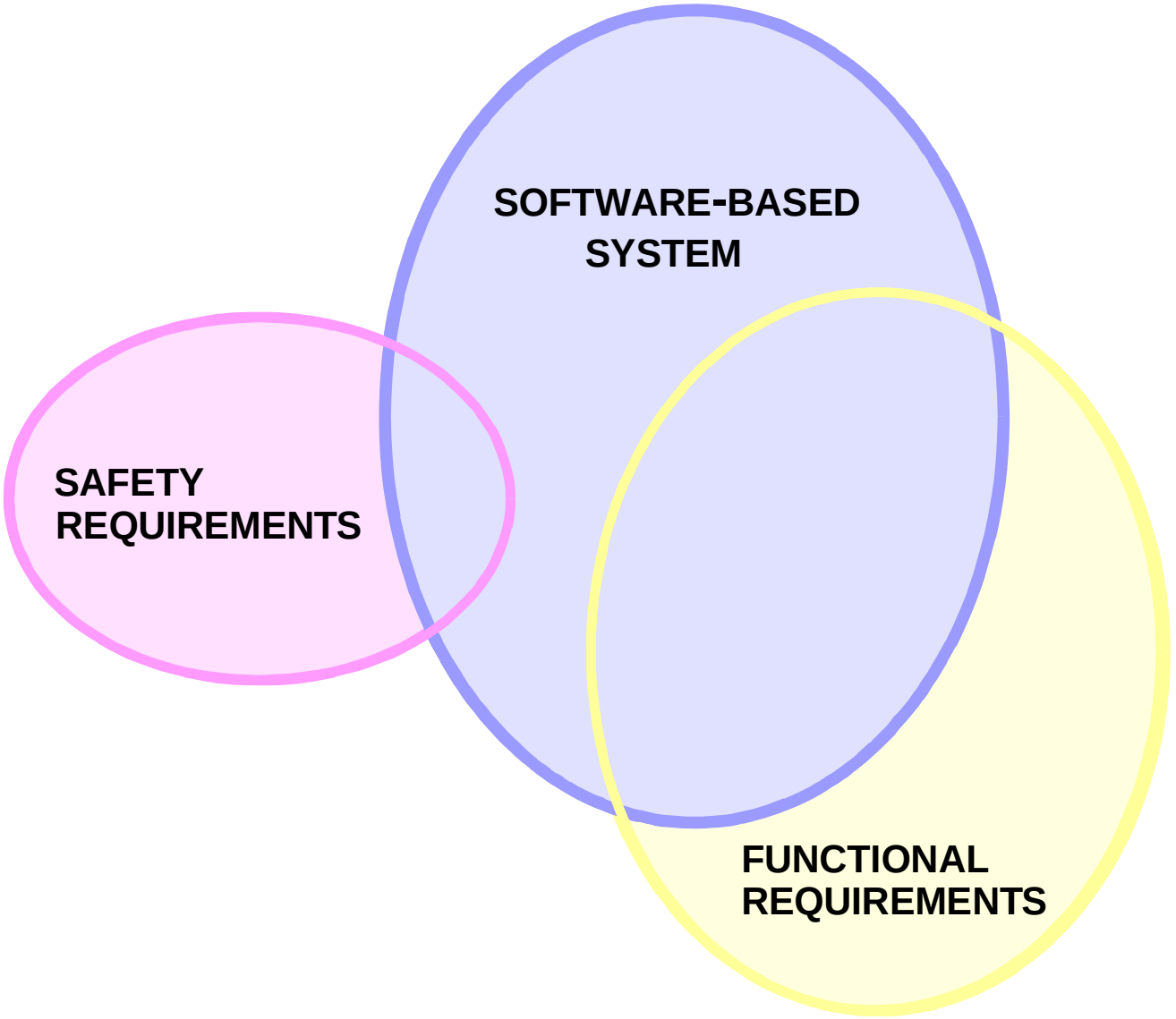
- **non-deterministic behaviour**
  - -> *Petri net: time-dependent dynamic conflicts*

- **dedicated testing techniques to guarantee reproducibility, e.g. Instant Replay**

Model Checking of Software-Based Systems


controller
environment
safety requirements
compiler
modelling
library
functional requirements
control model
environment model
composition
temporal logic
system model
set of temporal formulae
errors / inconsistencies
MODEL CHECKING

OBJECTTIVE - REUSE OF CERTIFIED COMPONENTS



- **GENERAL PROPERTIES (REQUIREMENTS)**

  -> *must be valid for any system, independently of its special functionality*

  -> *boundedness*
  -> *liveness*
  -> *reversibility*
  . . .

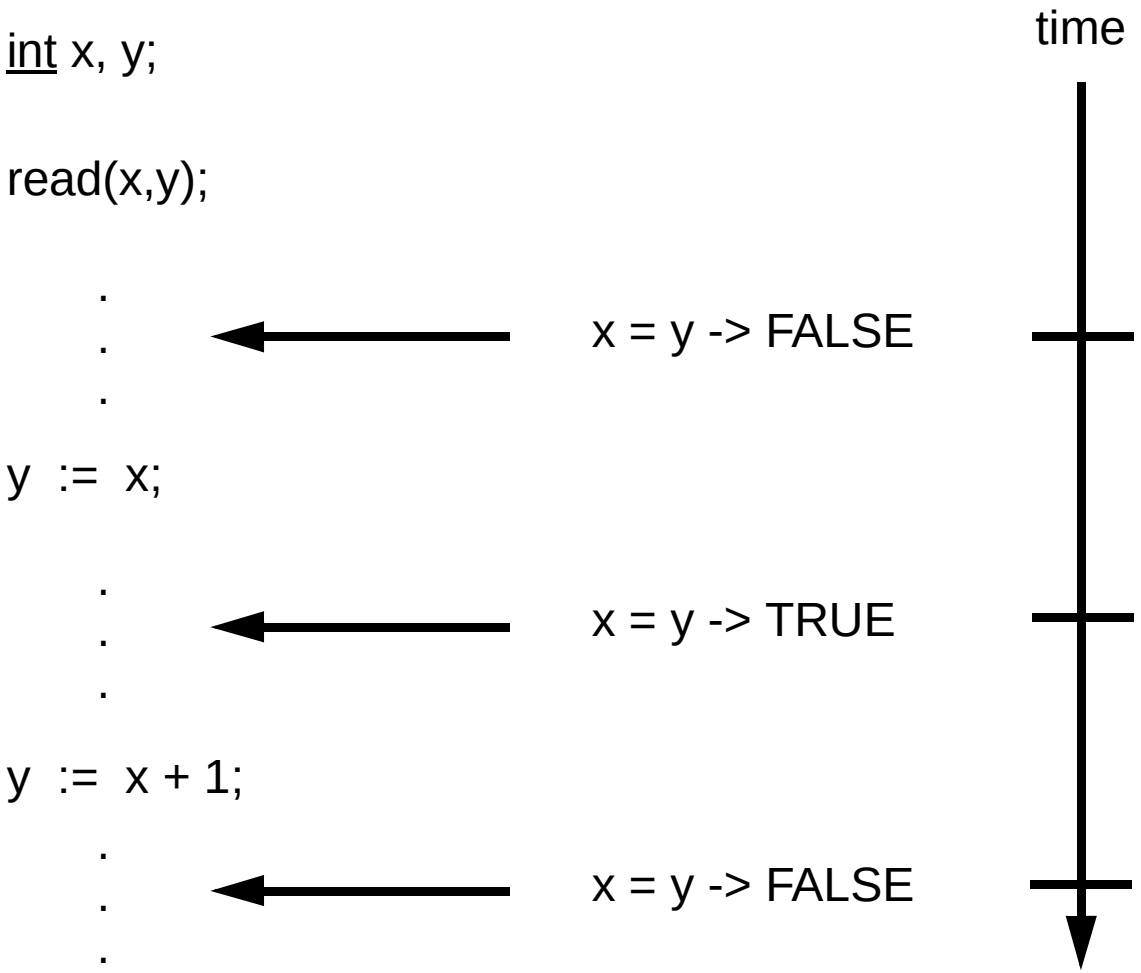
- **SPECIAL PROPERTIES (REQUIREMENTS)**

  -> *reflect the special functionality*

  -> *(i) insights into system behaviour*
  -> *(c) consistency propertiesto check the model's integrity*
  -> *(s) safety properties - "something bad never happens"*
  -> *(p) progress properties - "something good will happen finally"*
  . . .

- **insights (i)**
  - -> *Is it possible that both robot arms hold a plate simultaneously ?*
  - -> *How many plates can concurrently be inside the system ?*
- **consistency (c)**
  - -> *The swivel is always positioned at exactly one angle.*
  - -> *At any time, a robot arm can be driven in one direction only.*
- **safety (s)**
  - -> *To avoid machine collisions, two adjacent pushers will never be moved at the same time.*
  - -> *The press will only be closed, when no robot arm is positioned inside.*
  - -> *The feed belt may only convey a blank through its light barrier, if the table is in its loading position.*
- **progress (p)**
  - -> *Any plate at the input position will finally reach the cell's output position.*

Temporal Logics



How to specify logical statements with respect to the execution of a system ?

- **extension of classical (propositional) logics by temporal operators**

- **atomic propositions**
  - -> *elementary statements, having - in a given state - a well-defined truth value*
  - -> *e. g. mutex,* *for 1-bounded Petri net* *(Boolean)*
  - -> *e. g. buffer = 2, buffer > 2,* *else* *(Integer)*

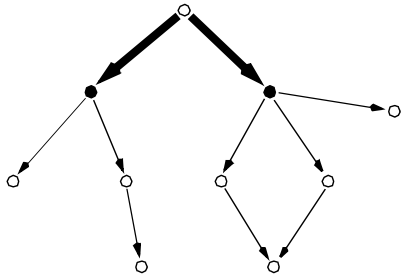
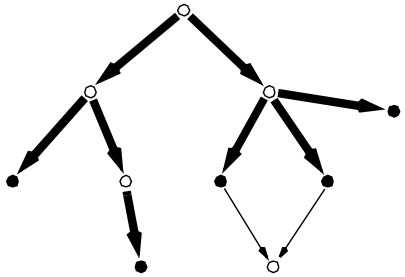
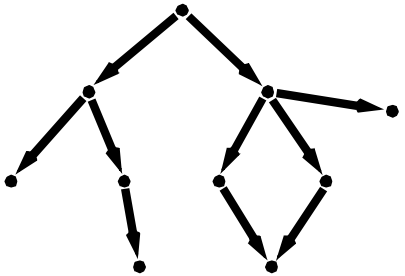
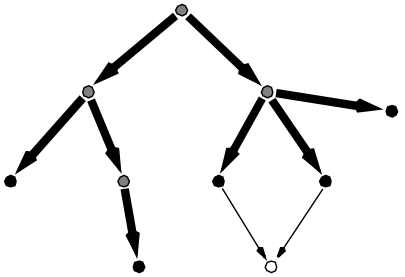
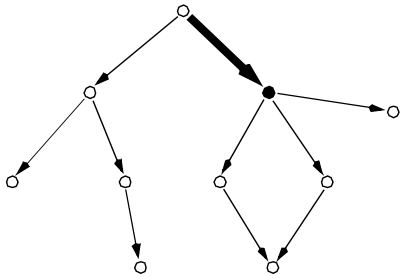
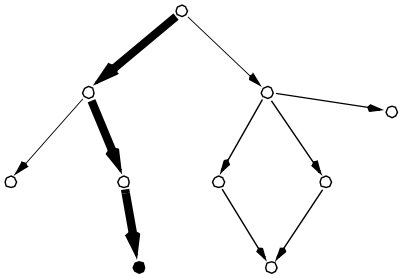
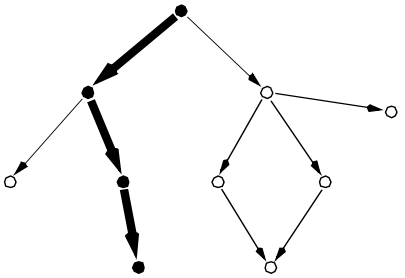
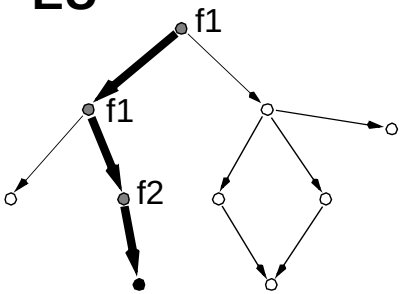
- **constants**
  - -> *TRUE, FALSE*

- **classical Boolean operators**

negation	*!*	*conjunction*	*
disjunction	+	*implication*	->

- **temporal operators**
  - -> *to refer to sequences of states*

CTL Operators, Interleaving Semantics

	next f	finally f	globally f	f1 until f2
on all paths	**AX** 	**AF** 	**AG** 	**AU** 
on path some	**EX** 	**EF** 	**EG** 	**EU** 

- **reachability-related** **EF (φ)**
  -> *There exists at least one computation path to reach eventually a state, where φ will be true.*

- **safety-related** **AG (! φ)** **-> equivalent to ! EF (φ)**
  -> *For every computation path, φ will never be true.*

- **invariant-related** **AG (φ)** **-> equivalent to ! EF (! φ)**
  -> *For every computation path, φ will be true for ever.*

- **liveness-related** **AG (EF (φ))**
  -> *What ever happens, there exists the chance (at least one path) that φ will be true again.*

- **progress-related** **AG (AF (φ))**
  -> *For every computation path, φ will eventually be true.*

- **insights / reachability**

  -> *Is it possible, that both robot arms carry a plate at the same time?*

  EF *(arm1_mag_on * arm2_mag_on)*

- **safety property**

  -> *If a robot arm is loaded, its magnet is not deactivated until the robot is in its unloading position.*

  AG *(φ -> **A** (!arm1_mag_off **U** ψ)), with*

  *φ = arm1_mag_on * arm1_pickup_angle * arm1_pickup_ext*

  *ψ = arm1_release_angle * arm1_release_ext*

  -> *The feed belt may only convey a blank through its light barrier, if the table is in its loading position.*

  G *(belt1_light_barrier_true -> (table_load_angle * table_bottom_pos))*

- **consistency property**

  -> *The swivel is always either stopped or moves in exactly one direction.*

  G *(robot_stop xor robot_left xor robot_right)*

Bottleneck:
State Explosion
(no primitive recursive function ...)

DINING PHILOSOPHERS 1000, BDD-BASED BY JSP

Number of places/marked places/transitions: 7,000/2,000/5,000

Number of states:

113751760865620516280672035436276768405854187694780001109285823216991 8\\
159959588122031332641120690971790713407413960379370132051412946235771 0\\
244289522738424241885324723952294300718880861927052755597203329394869 1\\
334498271287409035878953318171137286359195790723689557093738307422542 1\\
493299735055934871120872608511650262781852464476299128123872281683542 6\\
439043702222222716712699874004961590120093014497021663026892511863169 6\\
792192797756430854076755677722422066045029462353435568315492194903488 7\\
413893510872611522753508464671945735340847108696533249480549775338294 2\\
171781101168772051021154169003921176627995642292903237688541475038527 5\\
512488192401053636525511904747774118 74

ca. 1.1 * 10^667

Number of places/marked places/transitions:	*7000/2000/5000*
Time to compute P-Invariants:	*45885.66 sec*
Number of P-Invariants:	*3000*
Time to compute compact coding:	*385.59 sec*
Number of Clusters:	*3000*
Number of Variables:	*4000*
Time:	*3285.73 sec, ca. 54.75 min*

POSSIBLE ANSWERS TO STATE EXPLOSION

- **compositional methods**
  -> *simple interfaces between modules*
- **abstraction by ignoring some state information**
  -> *conservative approximation*
- **different logics -> different algorithms, e.g. LTL**
- **integer programming**
- **compressed state space representations**
  -> *BDD, IDD, ...*
- **lazy state space construction**
  -> *partial order methods*
- **alternative state spaces**
  -> *partial order representations, e.g. prefix*

'BASE CASE' TECHNIQUES

'ALTERNATIVE' METHODS (PN)

TEMPORAL LOGICS, OVERVIEW

semantics / time	interleaving	partial order
linear (LTL)	**traces** (no conflict, no concurrency) Manna & Pnueli, Kröger, jsp 2001 ***BDD/IDD-LTL, ...***	**runs** (no conflict, but concurrency) Reisig *tools: ?*
branching (CTL)	**reachability graph** (conflict & concurrency not distinguishable) Emmerson, Clarke *PROD,* ***Charlie, BDD/IDD-CTL, ...***	**prefix** (conflicts & concurrency) McMillan, Esparza, pd 2001 *PEP*

TECHNIQUES & TOOLS, OVERVIEW

technique	CTL	LTL
reachability graph	INA, Charlie	PROD, MARIA
stubborn set reduced reachability graph	LoLA	PROD (LTL\X)
symmetrically reduced reachability graph	LoLA (symmetric formulas)	?
BDD, NDD, ..., xDD	BDD-CTL, SMART, IDD-CTL	BDD-LTL
Kronecker algebra	[Kemper]	?
prefix	PEP (CTL_0)	QQ (LTL\X)
process automata	[pd]	?

requirement formula	# states generated	time effort
8 a	2259	4.16’
8 b	1775	3.76’
9	2305	4.34’
10	1879	3.10’
15	1184	2.55’
29	704	2.16’
30	703	2.46’
31 a	27104	23.80’
31 b	6433	4.02’

-> control model, 5 plates, full state space: 1,657,242 [7,185,779]

STATE SPACES, EXAMPLES - BDD/IDD

case study	net size	# of states
production cell, 5 plates	231 P / 202 T	1.6 * 10^6
production cell, 3 plates		7.1 * 10^6
1000 dining philosophers	7,000 P / 5,000 T	1.1 * 10^667
solitaire, standard		
solitaire, non-standard		
Halobacterium, motor 44		
MAPK cascade, 80 levels		
MAPK cascade, 120 levels		
gene regulation, 6 cells, m01		
gene regulation, 6 cells, m02		

Example - GAMES

SOLITAIRE

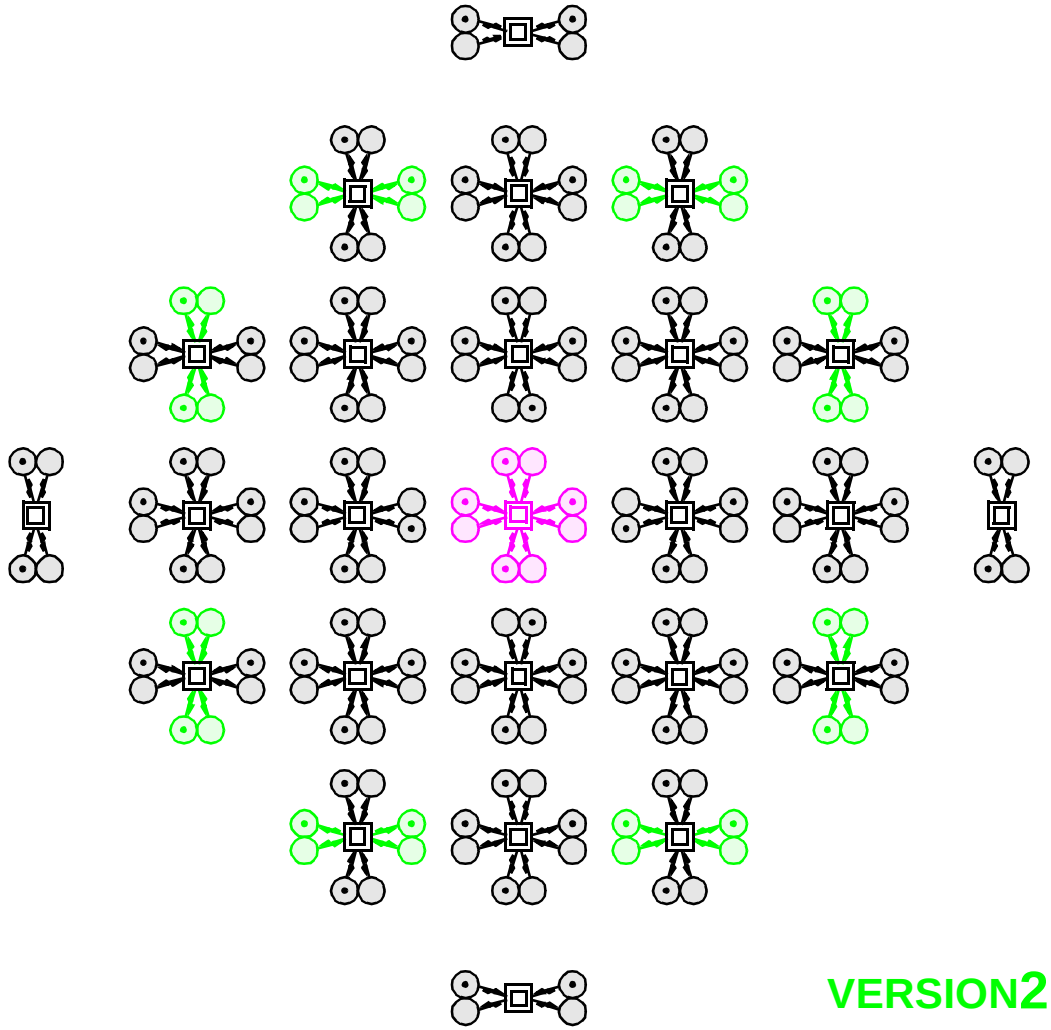
- two versions,
  green squares Y/N
- all but one squares
  carry tokens
- remove tokens
  by jumbing over them
- goal of the game:
  only one token left
- questions:
  is there a solution ?
- always ?

11	12	13	14	15	16	17
21	22	23	24	25	26	27
31	32	33	34	35	36	37
41	42	43	44	45	46	47
51	52	53	54	55	56	57
61	62	63	64	65	66	67
71	72	73	74	75	76	77

Example - GAMES

SOLITAIRE

- two-level hierarchical pn
- only one square net component
- two states for each square i: T(i), F(i)
- goal of the game: dead state(s) with Σ T(i) = 1
- reachable ?
- for any initial marking ?

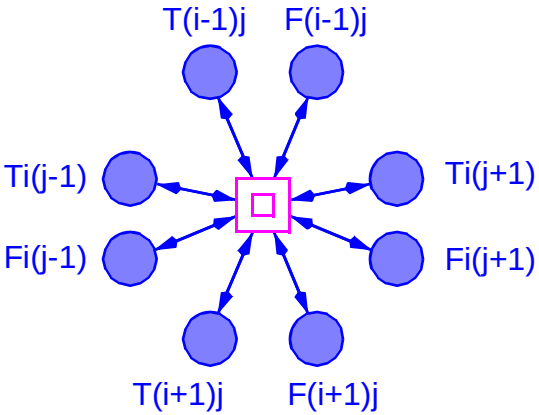


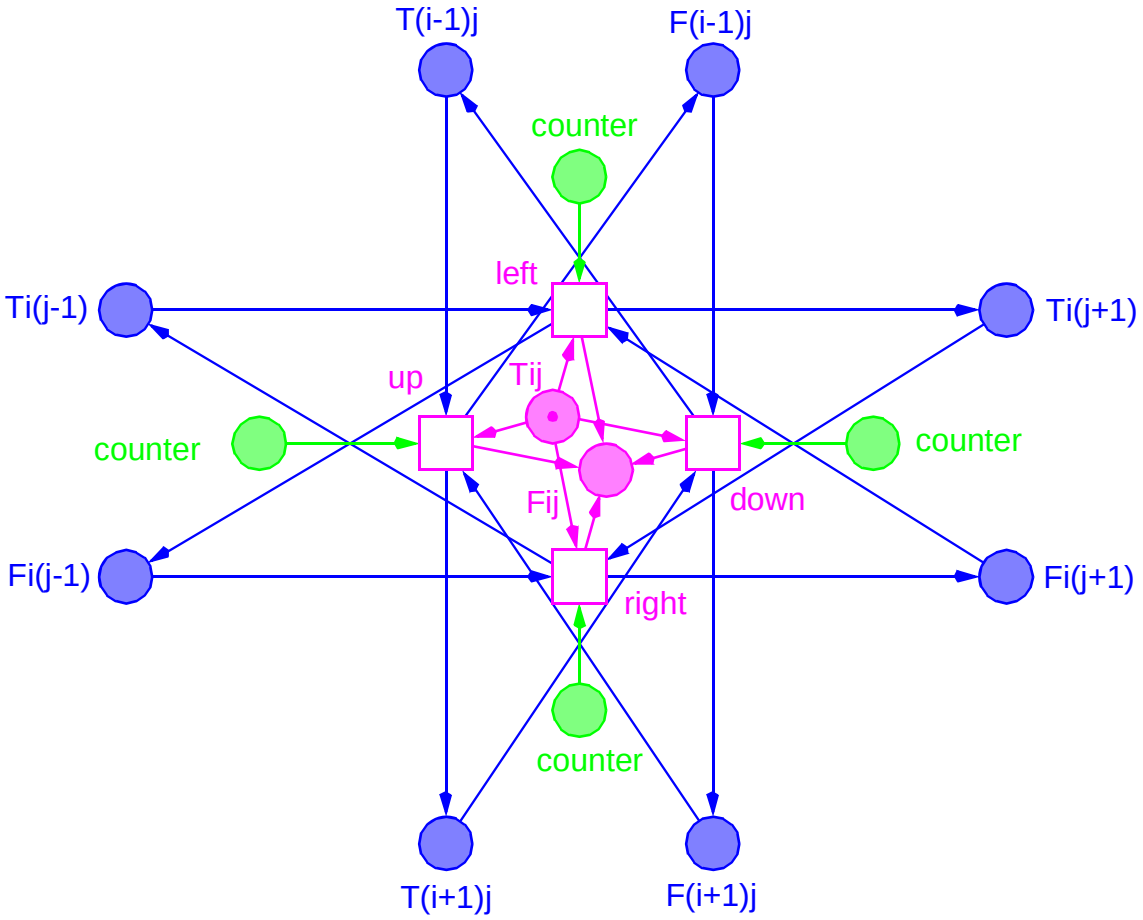
VERSION2

Example - GAMES

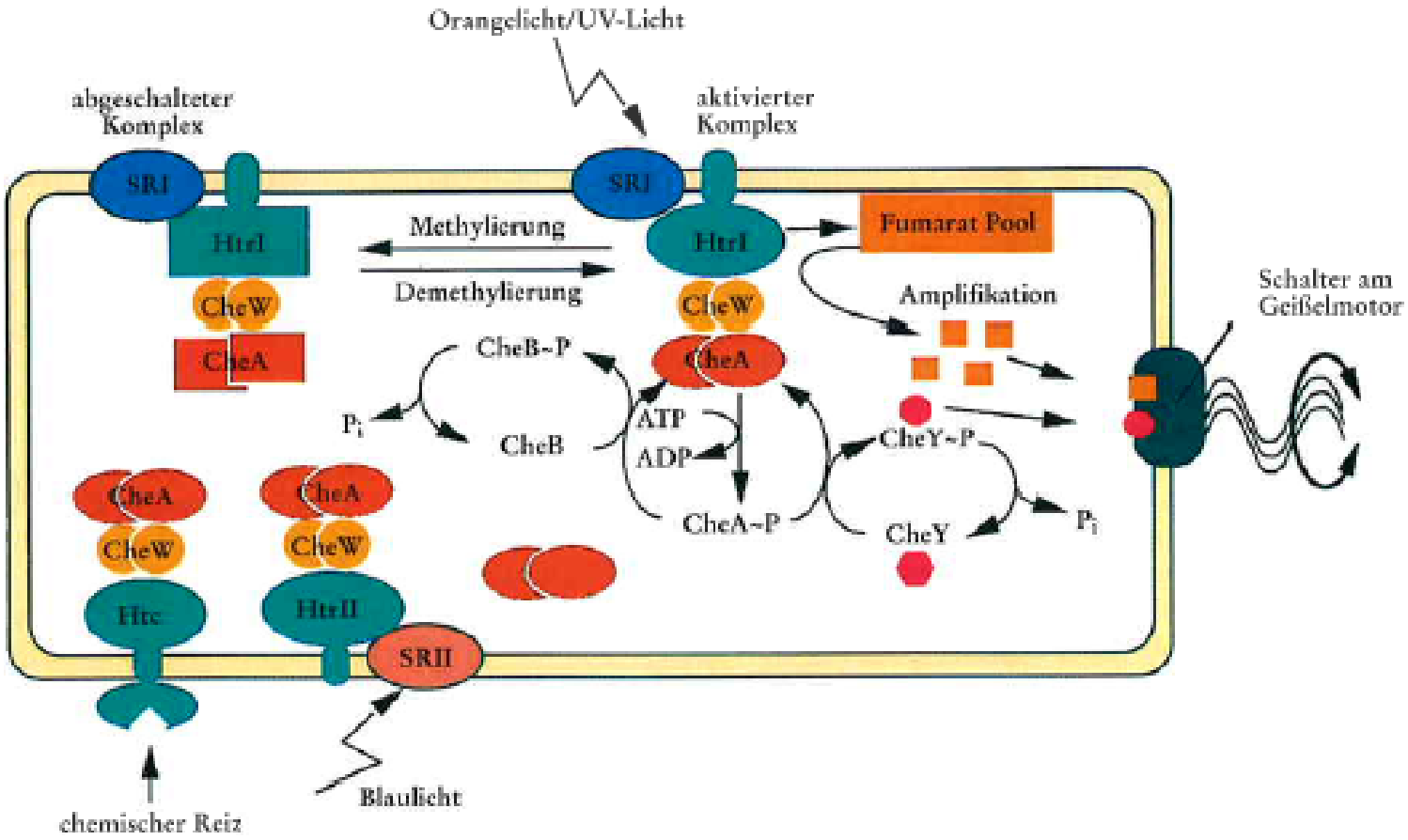
SOLITAIRE

- square component
- counter facilitates reachbility question, but hinders analysis



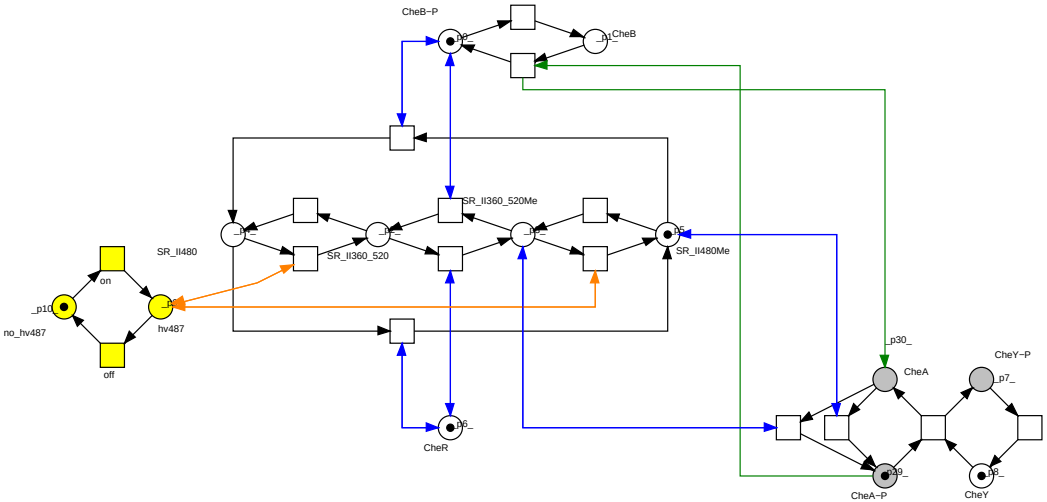


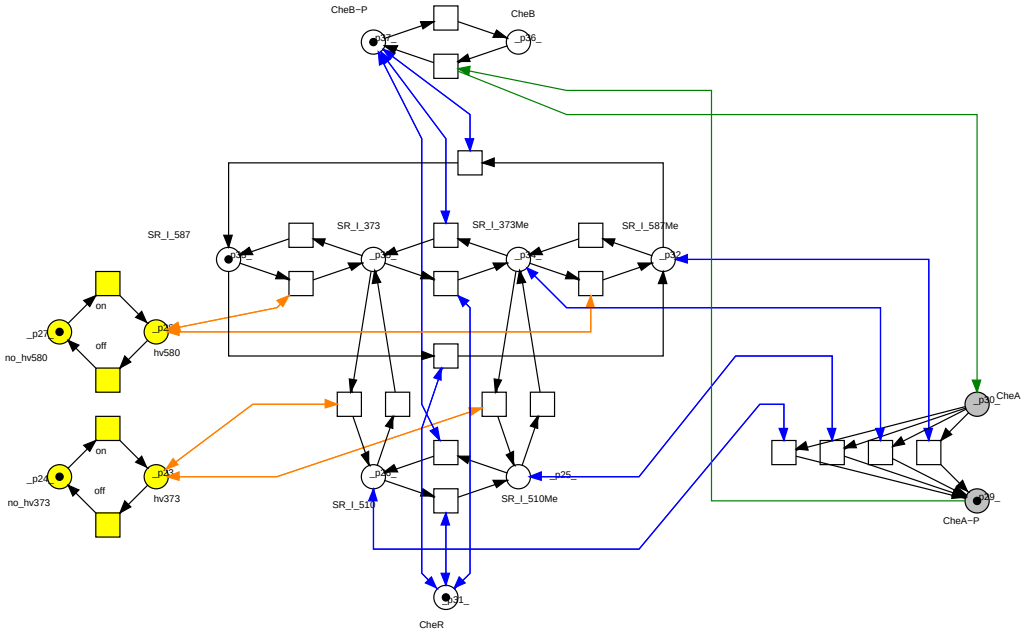
EXAMPLE - SWITCH CYCLE HALOBACTERIUM SALINARUM

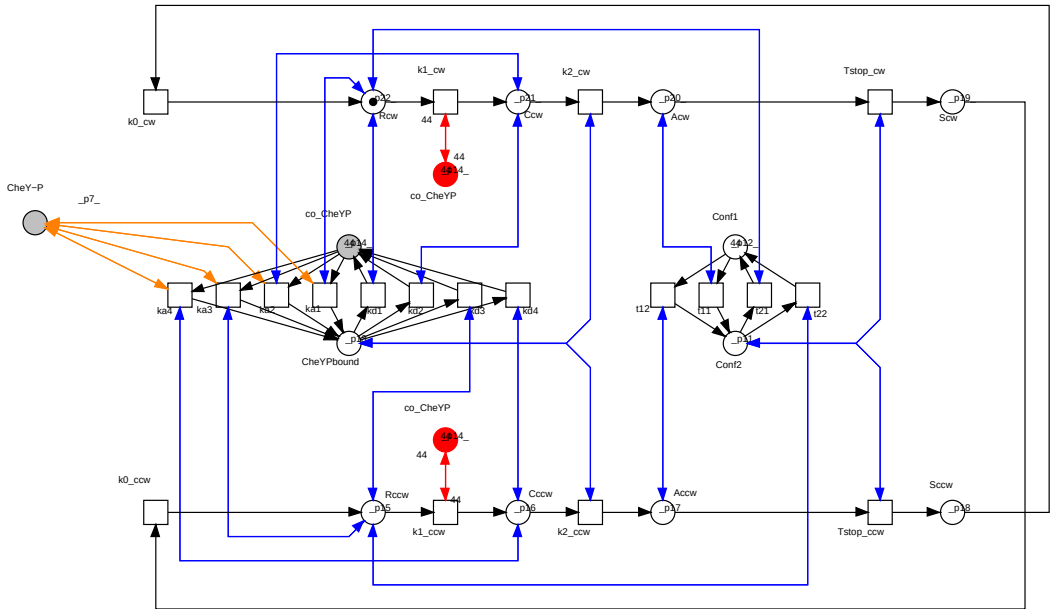


[Marwan; Oesterhelt 1999]

EXAMPLE - SWITCH CYCLE HALOBACTERIUM SALINARUM



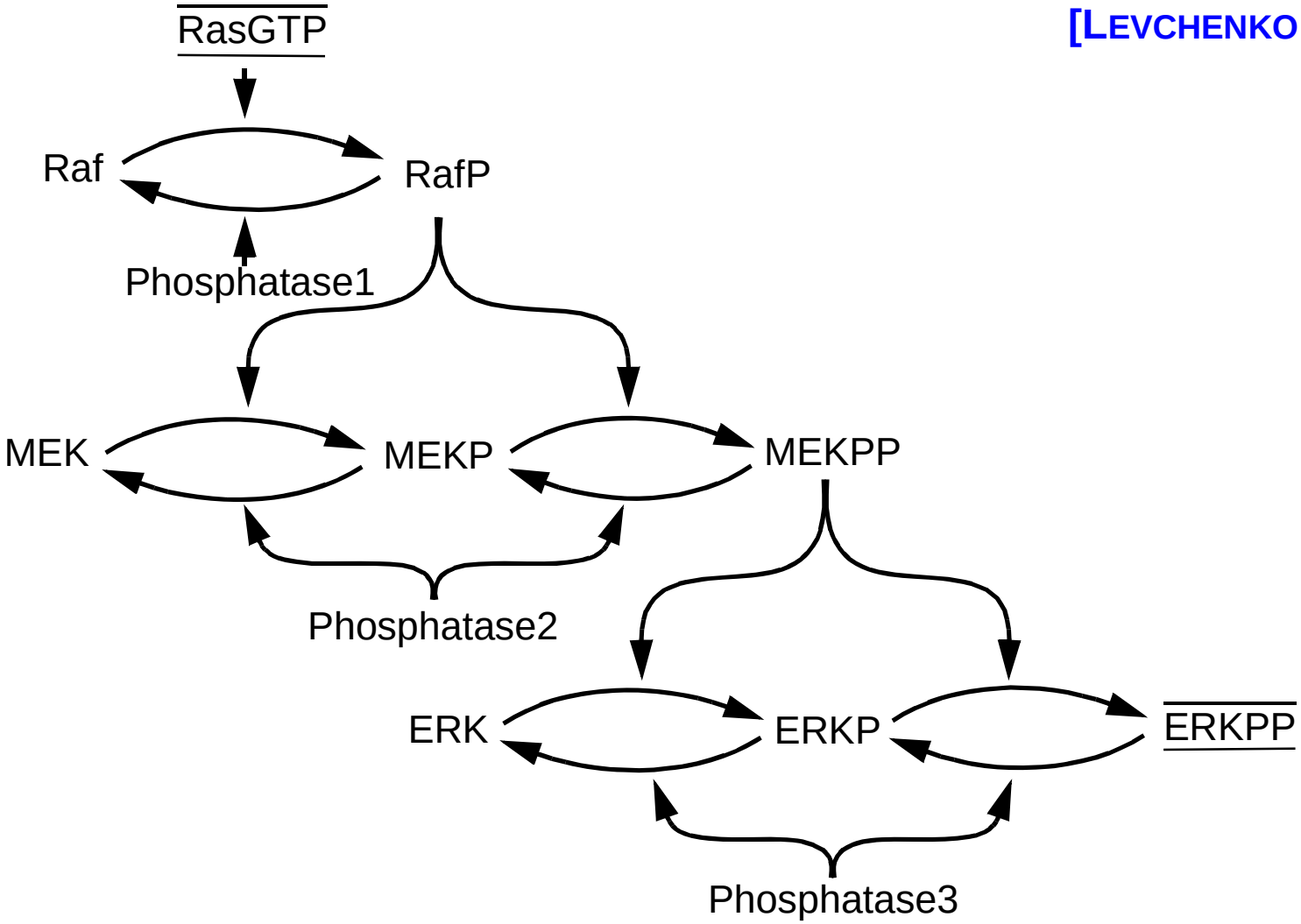




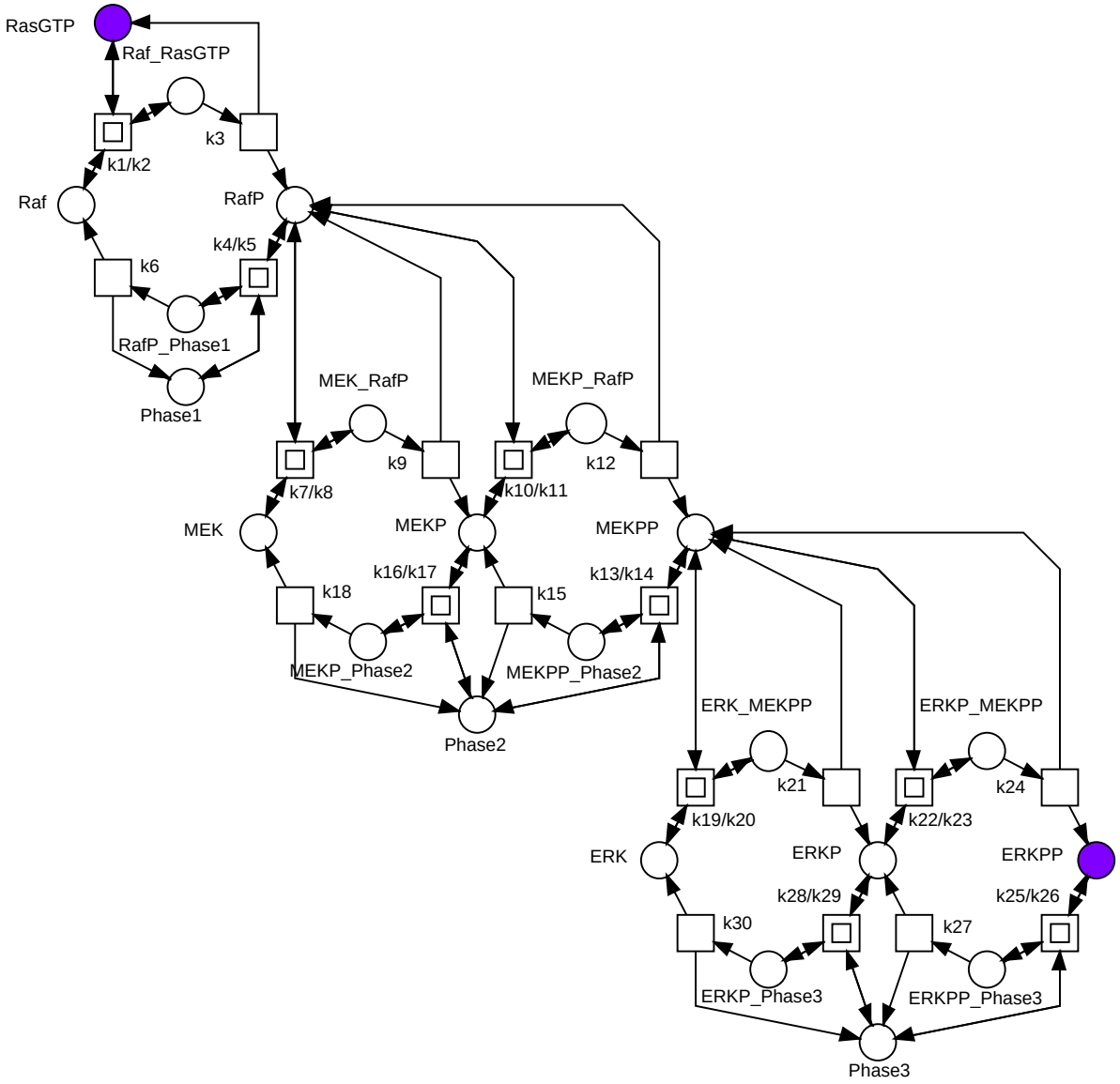
[UNPUBLISHED]

EXAMPLE - MAPK SIGNALLING CASCADE

[LEVCHENKO 2000]



EXAMPLE - MAPK SIGNALLING CASCADE



[GILBERT, HEINER, LEHRACK 2007]

[HEINER, GILBERT, DONALDSON 2008]

STATE SPACES, EXAMPLES - BDD/IDD

case study	net size	# of states
production cell, 5 plates	231 P / 202 T	1.6 * 10^6
production cell, 3 plates		7.1 * 10^6
1000 dining philosophers	7,000 P / 5,000 T	1.1 * 10^667
solitaire, standard	65 P / 75 T	1.8 * 10^8
solitaire, non-standard	73 P / 91 T	2.9 * 10^9
Halobacterium, motor 44	38 P / 60 T	4.8 * 10^7
MAPK cascade, 80 levels	22 P / 30 T	5.6 *10^18
MAPK cascade, 120 levels		1.7 * 10^21
gene regulation, 6 cells, m01	144 P / 307 T / 2146 A	7.4 * 10^21
gene regulation, 6 cells, m02		3,3 * 10^25

SUMMARY MODEL CHECKING

- **correct model checking requires correct requirement specifications**
  - -> *take your time, think twice*
- **model checking proves consistency of**
  - -> *model & specification*
- **if the answer is NO (-> FALSE)**
  - -> *the model can be wrong*
  - -> *the specification can be wrong*
  - -> *or both*
- **if the answer is YES (-> TRUE)**
  - -> *model & specification can still contain (same) logical faults*
  - -> *unlikely !?*
- **Model checking is no substitute for thinking !**

SUMMARY SOFTWARE VALIDATION

- **validation can only be as good as the specification**
  - -> *readable <-> unambiguous*
  - -> *complete <-> limited size*
- **validation is extremely time and resource consuming**
  - -> *'external' quality pressure*
- **sophisticated validation is not manageable without theory supported by tools**
- **validation needs knowledgeable professionals**
  - -> *study / job specialization*
  - -> *profession of "software validator", "software tester"*
- **There is no such thing as a fault-free program !**
  - -> *sufficient dependability for a given user profile*
- **Validation is no substitute for thinking !**

THERE IS NO SUBSTITUTE FOR THINKING !

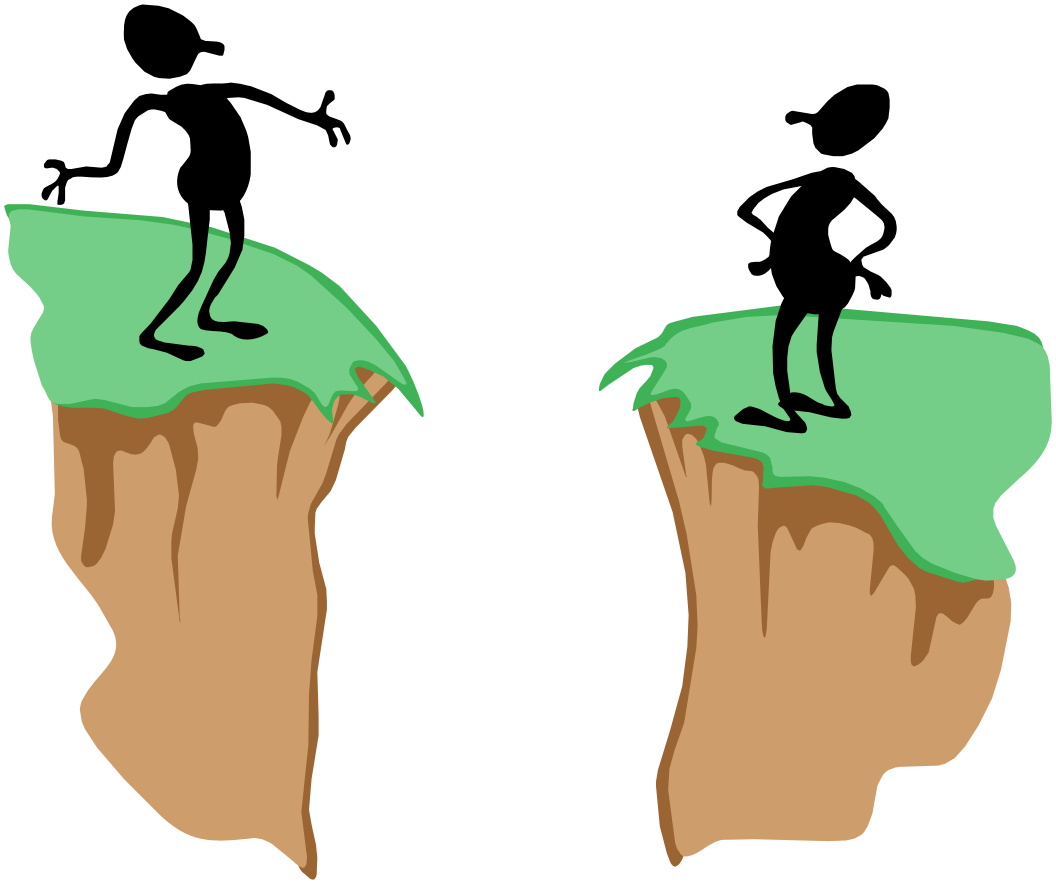
REFERENCES

- E.M. Clarke, O. Grumberg, and D.A. Peled:
  Model checking;
  MIT Press 1999, third printing, 2001.

- M. Heiner, P. Deussen, S. Spranger:
  A Case Study in Design and Verification of Manufacturing Systems with Hierarchical Petri Nets;
  Journal of Advanced Manufacturing Technology (1999), 15, pp. 139-152.

- C. Schröter, S. Schwoon, and J. Esparza:
  The Model Checking Kit.
  In Proc. ICATPN, LNCS 2679,Springer, 2003, pp. 463-472.

- A. Tovchigrechko:
  Model checking using interval decision diagrams;
  PhD thesis, BTU Cottbus, Dep. of CS, 2008.



Thanks !

http://www-dssz.informatik.tu-cottbus.de