Formale Methoden in Software Entwurf

Verification ⇔ Validation
- Impl = spec
- requirements as intended

Defects

dangerous
expensive
hard to patch

: Digitalisation
↗ security critical systems

⇒ prevent

+ confidence in system

civil engineering

calculations/estimations
Hardware redundancy
robust design (no single point of failure)
Seperation of subsystems
Design patterns

but

bit-flip changes behavior

complexity underestimated, non-continuous func.
redundancy as replication same bug
signals propagate to whole system
Properties not compositional
untrained engineers, immature designs
time-to-market > reliability
cost

Formal

mathematics, symbolic logic
formality basis for tools

check if impl. satisfies spec — no validation

+ dev time
- gen test cases
- support debugging

Testing

against internal SW errors (bugs)
external faults by simulating them and tracing propagation

inputs
- fuzzing - no guarantees
- chosen - expensive by hand
- automatic - need spec/model

test coverage
test oracle = spec
↳ desirable

- can only show presence of bugs
- labor intensive

formal methods can partially replace testing
+ potentially validating

Specification

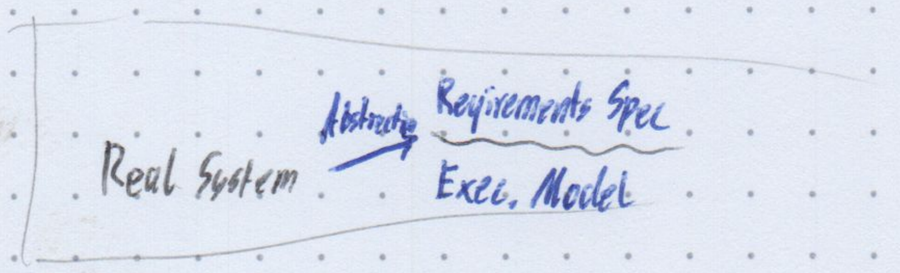
Simple properties
Safety " : something bad will never happen
"Liveness" : " good " happen eventually

Schematic "
Deadlock-freedom, no starvation, fairness
Reachability, no uncought except./errors

Non-functional "
Runtime, Mem, Useability

Full behavioural spec
impl. satisfies contract
data consistency/system invariants
Modularity, Encapsulation
Program equivalenz e.g. compiler correctness

Relational properties
refinement, information flow, fault propagation



Proving system properties

Abstract ↔ Concrete level (model)

e.g. finite automaton
+ automatic proofs
- unfaithful modeling inevitable
may capture essence (e.g. of protocols)

e.g. JAVA/C program
- unbounded data types
complex
in general not automatic proofs
precise characterization of semantics

Simple ↔ Complex properties (Spec)

simple, schematic properties
finite case distinctions
approximation propositional logic
automatted proofs

full behavioural spec
quantification over infinite domains
first order logic

e.g. SPIN — KeY

Automatic ↔ Semi-Automatic Proof

batch/auto-active mod
may fail or be inconclusive
tune param
patch program, spec

interactive
intermediate inspection
proof checked by tool
— needs knowledge of tool

Hardware ↔ Software Verification

good match
limitations of tech and application

specialized software
check of abstraction not code directly

Pitfalls formal models

Creation

(Abstraction) e.g.
over-simplified — no latency
missing requirement — max length
incorrect model — int - float

machine checkable
declare signatures (symbols) to spot incompleteness

Errors give deep insights

BRUNNEN

Deductive Verification
proofing spec Conformity

e.g. for
Hardware Verification
(FPGA)
Software
safety critical
parts of OS
...
Libaries
JDK LinkedList
...
tools
Frama-C
...

SPIN verify Promela model against temporal logic specs
- meaningfed properties

KEY Java against JML contracts
- proof size
finding loop invariant

=> Tool supper
for repetitive tasks
+ avoid clerical errors
+ cope large programms
certifiable

free, well documented

PROMELA – Process meta-Language

based on ω-automata, linear temporal logic

Exec
spin ...pml
ispin ...pml

modeling
- finite state space
- pure expressions
- few control structs
- Multi-threaded
- shared Mem
- Sync & Message passing

not programming language
- no floats
- encapsulation
- methods
- ref. types
- API, Libary
- input, GUI

but Non-deterministic
scheduling policy configurable

∀ init mit 0

arithmetic Data types

- int
- short, byte
- bit = {0,1} = bool

unsigned

Ops +, −, *, /, %

computed as int then converted

Arrays

static constant

type name[N]
- only 1-dim
- no a==b – always diff types

" name[5]

Enumerations

mtype[:tag] = { name, ... }; // multiple declarations ⇒ union of them
max. 255 literals

mtype[:tag] var = name;

Records

typedef NAME {
  type attr, ...;
  ...
};

NAME a;
a.attr...

only previous declared records ⇒ no self-references

multi-dim arrays
typedef VECTOR { int vector[10] };
VECTOR matrix[5];
...

Control Statements

Sequencing

; seperator

conditional expr. atomic
(boolean-guard → then : else)

Guarded Commands

not atomic

if
:: <guard-statements>+ -:: bool exp | else → ; command
fi | od

- ≥1 true : exec one arbitrary
- =0 true : block until one true

can lead to timeout

only way to exit
break / goto

Syntactic sugar
for (i : 1 .. N) { ... }
for (i in array) { ... }
i durch indizes

Jumps

goto Label – unique per process
=
Label: // not in front of guards

Inlining

inline name(a, b, ...) { // dekl. before use
  ... // new scope
}
=
name(...)

anonymous var –
+ no state
+ ~~vers~~übersicht

Non-Deterministic Assignment

if
:: range = 1
:: range = 2
...
fi

do
:: range < HIGH → range++
:: break
od

not equal chance
but doesn't matter
for verification

select(i : LOW .. HIGH)

printf("... %d", decimalvar)

Concurrent Processes

```
active prototype Name(){
  //code
}
active prototype Name(){
  //code
}
```

up to 255 uniquely named Processes

executed interleaved on single processor

```
prototype Name (type name, ...){...}
```

type name: global / local-in P

Sets of Processes

```
active [N] prototype P(){
  // ... _pid //own local vars
}
```

State of Process

program counter
var values

Computation with $P_1, \dots P_n$

$c^i = (s_0^i, \dots)$ $i \in (1, \dots n)$

time →

s_0^i : start of P_i
s_{j+1}^i : successor of s_j^i

Terminating
$\forall c^i$ finite $\wedge$ final state reached (end of code)

Blocking
$\forall c^i$ finite $\wedge \exists i$: final state not end of code

Infinite
$\exists c^i$ infinite

End locations of process
textual end
label end×××:

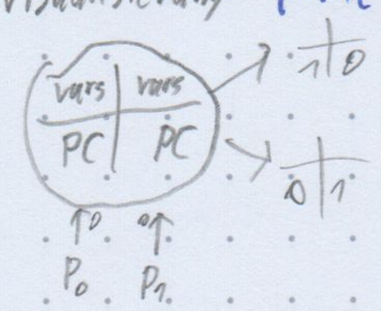
finite domains
fixed #processes
finite #statements

Interleaving

$(s_0, s_1, \dots)$ interleaving of $(c^1, \dots c^n)$

$\Rightarrow$ $s_j = s_{j'}^1, s_k = s_{k'}^1, j < k \Rightarrow j' < k'$
$\wedge$ $0 \le r \le k \Rightarrow s_j^i$ in interleaving

Visualisierung: finite directed graph



Weakly / Strongly atomic seq.

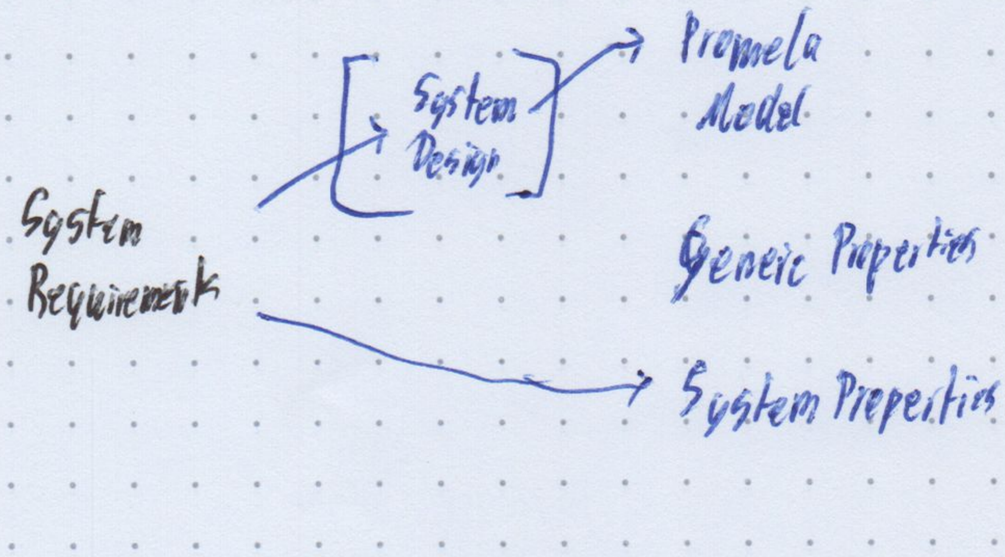
atomic {...} ⚠ not first (guard) blocking stmt ⇒ interleaving

d_step {...} never interrupted
- non-determinism always takes first option
- only first stmt may block, else error (guard)

Usage

Model essential

abstract complex computations (non-deterministic choices)
finite approx for unbound data structures
assumes fair scheduler



Select properties

generic: mutex, deadlock, starvation
system-specific: event sequences ...

Verify

$\forall$ possible runs sat properties
$\Rightarrow$ counter example

BRUNNEN

Verification

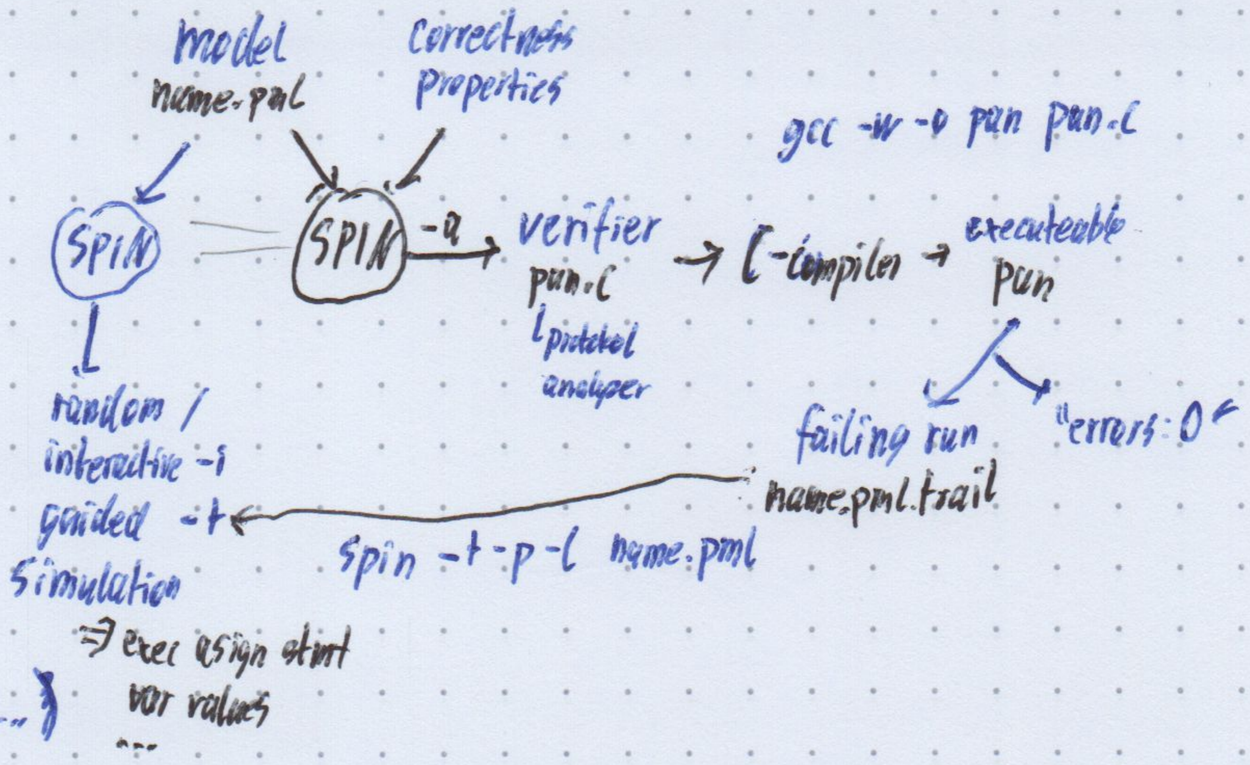
Simulation : one run

Verification
MC - Model Checker
∀ runs ∀ properties
exhaustive counter example search

R_M = Menge ∀ possible runs Model M

$R_{M,C_i} \subseteq R_M$ satisfying C_i

M correct Relative $C_1, \ldots \Leftrightarrow R_M = \{R_{M,C_1} \cap \ldots\}$



Stating Correctness Properties
outside model
never claims
temperal logic formulas
within model
Mete labels
end, accept, progress
Assertion statements — no quantification
⇒ z.B. GCD not fully verifiable
but sanity check
assert(expr)
└ if ≠0 ⇒ error

Modellierung
Seperation of concers
system ↔ property
verify should have not allready has

Richness ↔ simplicity

Concurrent / distributed systems

- combinatorial explosion
- data races, counts measured: deadlocks
- limited control (scheduling, speed
  performance, reliability communication)
- Testing - controllability ⇒ miss failures
  - reproduceability
  - resources for exhaustive

Initial Process pid=0

declared starts all p.'s
implicitly using »active«
explicitly

```
init {
  ... //z.B. starting p's
}      init global vars
```

run P() - starts copy of P drüber deklariert
(in atomic: only start exec next stmt after atomic)

(_nr_pr == 1) → ... - waits until its last p. running
└ # running processes

shared mem

⇒ global var
res ⇒ global var of bool / Enum
communication ⇒ global var of chan (channel)

Synchronization

no semaphores, locks, monitor primitives

Executability ⟺ blocking

assignments, assertions, print stmts	always
expr. stmt	iff ≠0
send / receive	
compound stmt atomic, d_step	iff guard
if, do	iff any alternative iff guard
for	always

process iff location counter

scheduler non-deterministic
non-blocking process

Critical Section CS

block of code accessing shared state

Problem Race Condition
⟺ result depends on exec order

Solution Mutual Exclusion ≤1 p exec it's CS at a time
! deadlock ∃p must eventually exec CS
! starvation ∃p''

Patterns

Busy waiting - wasteful

```
enterCritical P = true
do
:: ! enterCritical Q → break
:: else → skip
od
```

Block

```
enterCritical P = true;
! enterCritical Q
```

! deadlock
⇒ test and set

```
atomic {
  ! enterCritical Q;
  enterCritical P = true
}
```

! realistic in real system?
(inefficient)

Variations
at most n P's in CS
⇒ counter, semaphores, ...

refined mutex conditions
diff CS's
readers don't exclude readers
FIFO queue for entering (semaphores)

Verifying Mutex

system global property but assert code local

⇒ Invariants in Temporal Logic

Ghost Variables
vars only for spec / verifying
z.B. ein critical counter counting p's in CS
}

Verteilte Systeme

- Nodes ~> process
- communication channels ~> type chan
- protocols ~> algorithm

Global ↔ Lokal channel je nach us. Referenz
- ∀p: can send/receive
- even if passed to one other

```
chan name = [capacity] of {type_1, ... type_n}
            ↑ referenz teil          ↑ message Tupel
```

capacity = 0
=> Rendezvous channel

cap := capacity > 0
=> Buffered (FIFO)
part of state => keep small

checking status

full(ch)	cannot negate these => avoid not/else
nfull	
empty	
nempty	

send:

```
name ! expr_1, ... expr_n
↑ chan   ↑ types match
```

receive

```
name ? expr_1, ... expr_n
↑ chan   | expr_i is value => only receives msg_i == expr_i
         | expr_i is var => var := msg_i
           eval(var) to interpriete as value
```

Random receive

```
name ?? expr_1, ... n
```
first matching msg in queue

copy without removal

```
name ? <expr_1, ... n>
```

Prefix Syntax msg

```
expr_1 (expr_2, ... n)   für expr_1, ... n
```

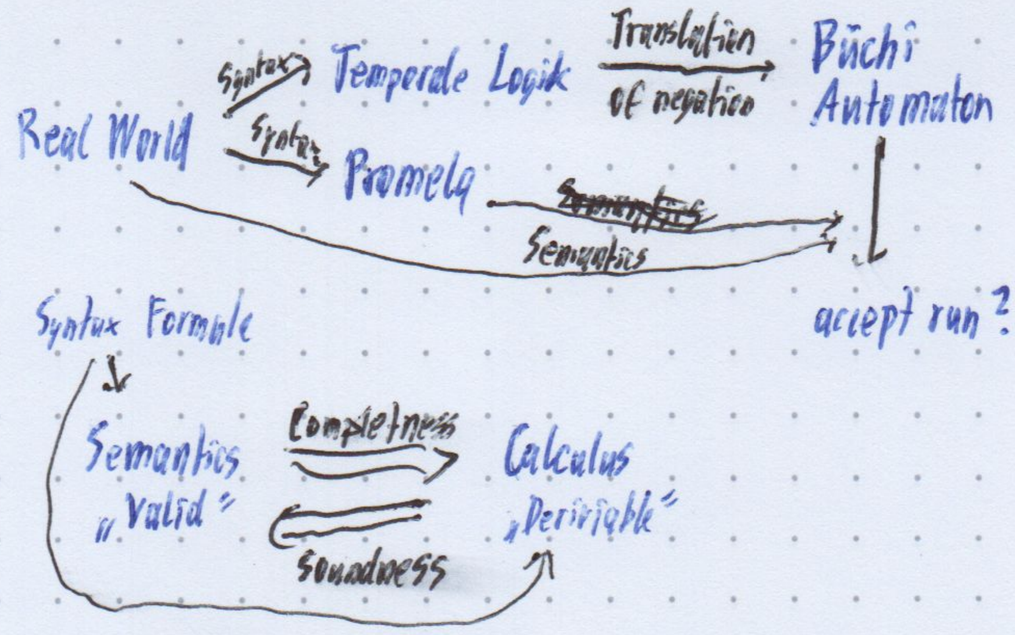
Executability
- Rendezvous
- Buffered async.
  - msg queue not full
- send/receive
- Synchron
  - « empty

Client-Server global req, reply channel
multiple servers => ! ensure responding to sender

=> sending _pid to server,
reply with it, pattern matching

=> sending local resp chan to server

Lineare Temporale Logik



Propositional Logic

Syntax

P - set of variables

Formulas For_0

true, false - Truth constants

Φ - vars connectives: Klammern verzichtbar

$\neg\phi, (\phi \wedge \psi), (\phi \vee \psi), (\phi \rightarrow \psi), (\phi \leftrightarrow \psi)$

in SPIN: ! && || -> <->

Semantics

Interpretation $I: P \rightarrow \{T, F\}$ ~ extension $\{q \in P \mid I(q) = T\}$

$val_I: For_0 \rightarrow \{T, F\}$ $val_I(p_i) = I(p_i)$ $val_I(true) = T$ ... $val_I(\phi \wedge \psi) = \begin{cases} T & \text{if } val_I(\phi) = T \text{ and } \dots \\ F & \text{sonst} \end{cases}$

$val_I(\phi) = T \Rightarrow I$ satisfies ϕ $I \models \phi$

$\forall I: (\forall \psi \in \Gamma: I \models \psi) \Rightarrow I \models \phi \Rightarrow \phi$ follows from Γ $\Gamma \models \phi$

Formalise Promela in Prob. Logic?

vars: e.g. byte n $\Rightarrow N_0, \dots N_7$ prob Vars

PC: PC_{ij} $\forall$ process i, line j

$\Rightarrow \Phi_{Program} := ((PC_{0,1} \wedge \neg PC_{0,2} \wedge \dots) \vee \dots) \wedge$ Startzustand, Übergänge

└ all reachable states $\mathbb{D} \models \psi \Rightarrow \psi$ true in $\forall$ reachable states

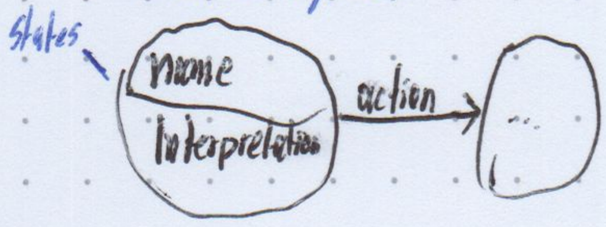
– cant express evolving state

e.g. n will become 0 eventually

changes value infinetly often ...

⇓ more expressiv Logic

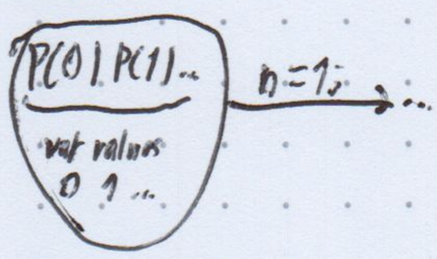
Transition Systems



Computations/runs - infinit paths

(finite stuck at final state)

Promela



$\mathcal{T} = (S, Init, \delta, I)$

- States
- initial states $\emptyset \neq Init \subseteq S$
- TransitionRelation $\delta \subseteq S \times S$
- Labeling $\forall s \in S$ I_s

run $\sigma = s_0 s_1 \dots$ mit $s_0 \in Init$ $\forall i: (s_i, s_{i+1}) \in \delta$

ϕ valid in $\mathcal{T} \Leftrightarrow \forall$ runs σ of $\mathcal{T}: \sigma \models \phi$

hard to automate

Temporal Logic extension of prob. logic

spec. sets of runs

ISPIN	Syntax		Semantic: sequence of interpretations
[]	$\Box\phi$	Always	Run $\sigma = s_0 s_1 \dots$ $\sigma_i = s_i s_{i+1} \dots$ (suffix)
<>	$\Diamond\phi$	Eventually	$\sigma \models true$ $\sigma \not\models false$
U	$\phi U \psi$	Until	$\sigma \models p$ iff $I_0(p) = T$ p∈P - in initial state
			$\sigma \models \neg\phi$ iff $\sigma \not\models \phi$
			...
			$\sigma \models \Box\phi$ iff $\sigma_k \models \phi$ $\forall k \geq 0$
			$\sigma \models \Diamond\phi$ iff $\exists k \geq 0$ $\sigma_k \models \phi$
			$\sigma \models \phi U \psi$ iff $\exists k \geq 0$ $\sigma_k \models \psi$ $\wedge$ $\forall 0 \leq j < k$ $\sigma_j \models \phi$

Validity $\models \phi$, ϕ valid $\Leftrightarrow \forall \sigma \in \{(2^P)^\omega$ - infinit seq. of interpretations$\}: \sigma \models \phi$

Represent set of runs e.g. $(\neg\phi\phi\phi\dots) \models \Diamond\Box\phi$

$\phi_0 \phi_1 \dots$ represents $\{s_0 s_1 \dots \mid I_j \models \phi_j \ \forall j \geq 0\}$

Properties z.B roulette mit ghost var

Safety - bad never happens $\Box$bad

Liveness - good eventually $\Diamond$good

infinetly often $\Box\Diamond\phi$

BRUNNEN

Regeln $\Box\Box\phi \leftrightarrow \Box\phi$ reflexiv

$(\neg\Box\phi) \Leftrightarrow (\Diamond\neg\phi)$ dual

$\Diamond\Box\phi \leftrightarrow (true\ U\ \phi)$

Formal Languages

alphabet/vocabulary Σ

word $w \in \Sigma^* = a_0 \dots a_n$

$L \subseteq \Sigma^*$ language over Σ^*

Formal ω-Language

ω-word $w \in \Sigma^\omega$

infinite seq. $a_0 \dots$

$L^\omega \subseteq \Sigma^\omega$ ω-language over Σ

Büchi Automaton over Σ

$B = (Q, I, F, S)$

Q locations/states $\neq \emptyset$ finite

$I \subseteq Q$ initial start locations $\neq \emptyset$

$F = \{F_1, \dots F_n\} \subseteq Q$ accepting locations

$S \subseteq Q \times \Sigma \times Q$ transition relation

in contrast to runs transitions not states labeld

Acceptance of $w = a_0 \dots$

$\Leftrightarrow \exists\, q_i \in S(q_i, a_i)$ $\forall i \geq 0$

$q_0 \in I$

$\exists f \in F$: infinetly often visited

$L^\omega(B) = \{w \in \Sigma^\omega \mid w$ accepted run of $B\}$

$L^\omega(B)$ accepted by B

ω-regular Language $\Leftrightarrow \exists B$ accepting language

ω-Regular expression

ab - a then b

$a+b$ - a or b

a^* - finitly often a

a^ω - infinitly often a

Theorem

Decidabifity whether $L^\omega(B)$ empty

Closure properties

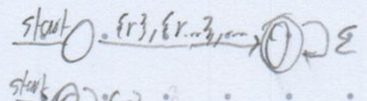
ω-languages closed under intersection, union, complement

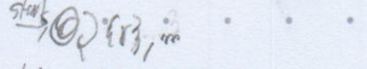
! deterministic Büchi less expressive

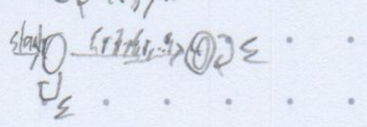
cannot accept all ω-expressions

LTL formula → Büchi Automaton

$\Sigma = \mathcal{P}(P)$ $I_{a \in \Sigma}(p) = p \in a \begin{cases} T & \text{if } p \in a \\ F & \text{const} \end{cases}$

$r \rightarrow$ 

$\Box r \rightarrow$ 

$\Diamond r \rightarrow$ 

Transitions System = Büchi Automaton

$\top \rightarrow B_\top$

$p := 1$ $p := 0$	p $\neg p$
start state	initial state
locations with end-label	accepting states
runs	words

BRUNNEN

Model Checking - gilt $\tau \models \varphi$?

check $h^u(B_\tau) \cap h^u(B_{\neg\varphi}) = \emptyset$

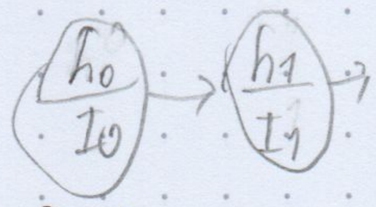
if not: send as counter example

Hies Temporal Logic = Linear temporal logic

Promela stating correctness properties

within model
- end labels endXXX:
- accept labels acceptXXX:
  - allows Acceptance Cycle
- assertions
  - + convenient always mutex, Array index bounds
  - + expressiveness
  - Absence Deadlock, Starvation
  - - no seperation of concerns
  - - easily out of sync (errors pone) forget
  - - be only state at their location

outside
- temporal logic
  - Boolean temporal logic
    - global vars, constants ~~of bool/bit $\in For_{BTL}$~~
    - of bool/bit $\in For_{BTL}$
    - numerical expressions
      - $e1 == e2, !=, <, <=, >, >= \in For_{BTL}$
    - Process@label $\in For_{BTL}$



h_j : Process → location
I_j : vars → value

$h_j I_j \models x < y \Leftrightarrow I_j(x) < I_j(y)$
$h_j I_j \models P@l \Leftrightarrow h_j(P) = l$ - e.g. for mutex
...

in file
- LTL <claim name> $\in$ {<formula>} ! ensure it/the right one is used
  - check use claim acceptance cycle
  - übersetzt in never claim $\neg\varphi \Rightarrow$ process with accept labels ! only one Process advances each step
    - accept ⇒ fail

acceptance cycle
- spin -a file.pml
- gcc -DSAFETY -o pan pan.c
- ./pan -N name

never { Process with accept labels }

$h^w(M) \cap h^w(NC_{\neg\varphi}) > \emptyset \Rightarrow \varphi$ holds in M

Weak Fairness check option spin -f

∀ continuously executeable stmt executed eventually

Strong Fairness

∀ oft executeable stmt exe eventually
harder to check not impl by most model checkers

First Order Logic

Signatur Σ for $TSym \neq \emptyset$ Set of types

- FSym set function symbols $f: A_1 \times .. A_n \to A$, $A_i, A \in TSym$ — Spezialfall ($n=0$) constant
- PSym predicate $p(A_1, ... A_n)$ — ϕ propositional variable
- VSym variable $v: A$

Syntax — type e.g. int
- type symbol (type, ..); ...
- symbol((type, ..); ...
- type symbol; ...

KeY \sorts{ a; ..; }

\predicates {
p(.., ..); ..; }

\problem{
//formula }

First order term of $A \in TSym$ over Σ Trm_A

$v \in VSym$ of A
$f(t_1, ... t_n)$ of A
└ typkorrekt

Semantic

Evaluation $val_{s,\beta}: Trm_A \to D^A$
$val_{s,\beta}(v) = \beta(v)$
$val_{s,\beta}(f(t_1, ..)) = I(f)(val_{s,\beta}(t_1), ..)$

universe / domain $D \neq \emptyset$ — $D^A = \{d \in D \mid \delta(d) = A\} \neq \emptyset$
type $\delta: D \to TSym$ — $D = \bigcup D^A$; $A \neq B \Rightarrow D^A \cap D^B = \emptyset$

Interpretation I of $f: D^{A_1} \times .. \to D^A$
of $p \subseteq D^{A_1} \times ... D^{A_n}$
of $\doteq = \{(d, d) \mid d \in D\}$ fixed

$S = (D, \delta, I)$
fo state/model

First order formulas Fml

Atomic formula
- true, false — $val_{s,\beta}(true) = T$
- $t_1 \doteq t_2$ equality — $val_{s,\beta}(p(t_1, ..)) = T \Leftrightarrow (val_{s,\beta}(t_1), ..) \in I(p)$
- $p(t_1, ... t_n)$ predicate

Var Assignment $\beta: VSym \to D$ respecting type

modified $\beta_y^d(x) = \begin{cases} \beta(x) & x \neq y \\ d & x = y \end{cases}$

$[x/t]\phi$ replace occurence var x with t

KeY-FOL

$\neg\phi$, $\phi \wedge \psi$, $\phi \vee \psi$ — !, &, | — $val_{s,\beta}(\phi \wedge \psi) = T \Leftrightarrow val_{s,\beta}(\phi) = T$ and ...
$\phi \to \psi$, $\phi \leftrightarrow \psi$ — ->, ...
$\forall x; \phi$, $\exists x; \phi$ — \forall A x; ϕ, .. exists — $val_{s,\beta}(\forall x; \phi) = T \Leftrightarrow val_{s,\beta_x^d}(\phi) = T$ for all $d \in D^A$, $x: A$ ($\exists$: for at least one)
└ x bound ↔ free
no free vars ⇒ formula closed

Semantic Concepts
- Satisfiability $val_{s,\beta}(\phi) = T$
- Truth $s \models \phi$ — ϕ true in s
- Validity $\models \phi$
- closed formula sat ⇒ true

close $\dfrac{}{\Gamma, \phi \Rightarrow \phi, \Delta}$ true $\dfrac{}{\Gamma \Rightarrow true, \Delta}$ false ... closeEq $\dfrac{}{\Gamma \Rightarrow t \doteq t, \Delta}$

notLeft $\dfrac{\Gamma \Rightarrow \phi, \Delta}{\Gamma, \neg\phi \Rightarrow \Delta}$ orLeft $\dfrac{\Gamma, \phi \Rightarrow \Delta \quad \Gamma, \psi \Rightarrow \Delta}{\Gamma, \phi \vee \psi \Rightarrow \Delta}$

andLeft $\dfrac{\Gamma, \phi, \psi \Rightarrow \Delta}{\Gamma, \phi \wedge \psi \Rightarrow \Delta}$ impLeft $\dfrac{\Gamma \Rightarrow \phi, \Delta \quad \Gamma, \psi \Rightarrow \Delta}{\Gamma, \phi \to \psi \Rightarrow \Delta}$

...Right

Semantic Argements
- uncountably states
- may infinite Domain
- at least exponential

allLeft / instAll $\dfrac{\Gamma, \forall_\tau x; \phi, [x/t']\phi \Rightarrow \Delta}{\Gamma, \forall_\tau x; \phi \Rightarrow \Delta}$ — t' varfree term

applyEqLeft $\dfrac{\Gamma, t \doteq t', [t/t']\phi \Rightarrow \Delta}{\Gamma, t \doteq t' \Rightarrow \Delta}$ applyEqRight $\dfrac{\Gamma, t \doteq t' \Rightarrow [t/t']\phi, \Delta}{\Gamma, t \doteq t' \Rightarrow \Delta, \phi}$

exLeft $\dfrac{\Gamma, [x/c]\phi \Rightarrow \Delta}{\Gamma, \exists_\tau x; \phi \Rightarrow \Delta}$ — c fresh constant

eqSymmLeft $\dfrac{\Gamma, s \doteq t \Rightarrow \Delta}{\Gamma, t \doteq s \Rightarrow \Delta}$

...Right

allRight.. , exRight/instEx

⇒ Syntactic Transformation
Sequent Calculus

Γ, Δ - set of formulas

Sequent: $\psi_1, ..., \psi_m \Rightarrow \phi_1, .. \phi_n$
Antecedent — Succedent
$\Leftrightarrow (\psi_1 \wedge .. \psi_m) \to (\phi_1 \vee .. \phi_n)$

Schema
$\Gamma \Rightarrow \phi \wedge \psi, \Delta$
main formula — side formulas
match top level occurance in Succedent

Soundness
closed proof ⇒ validity
Completness
valid formula ⇒ closed proof

Theorem sequent calculus is both

Rule

Premisse
Rule Name $\dfrac{\Gamma_1 \Rightarrow \Delta_1 \quad .. \quad \Gamma_r \Rightarrow \Delta_r}{\Gamma \Rightarrow \Delta}$
Conclusion

BRUNNEN

Proof tree structure, goal as root
closed ⇔ ∀ branches closed ⇐ rule without Premisse
durch ∀ ?

~~JML - Java Modeling Language~~

~~Fokus~~ Fokus: Unit Spezification
- interfaces
- classes
- methods
  - res. val.
  - initial val of formal param
  - accessible part of pre-/poststate

Design by
Spezification as Contract between caller & callee — Application / Library

Set of pre/post condition pairs
satisfied ⇔ (called in state sat. prec. ⇒ terminating ⇒ postcondition
- normal
- abrupt (exception)

JML - Java Modeling Language

Java Extension in comments

```
/*@ ... ; comment
  [@] ... ;
  [@] */
```

Spezification cases

~~/*@ ... behavior~~

```
/*@ public ... behavior
    [requires expr.;]*          mehrere ∧
    ([ensures expr.;]* - no Except thrown
    |[signals_only Exc1, ..., Excn;] - thrown Exceptiontype
     [signals (Exctype) expr.;]*) - post state depending on
    [diverges true;] - may not-termination   [assignable location, ....;] - frame condition
                                              \nothing
                                              \everything
                                              array[0..h]
    [also next case]*/
```

~~Data~~ Modifiers

```
/*@ spec_public @*/ - in ~~data~~ Definition public for spec   (auch mehrere hintereinander ↓)
    pure           - method: no sideeffects, always terminates
    nullable
    non_null       - default   bei array direct auch Elemente
```

Invariant - global state constraints

```
/*@ public invariant expr. */
    [static] ↔ Instance Invariant
```

- anywhere in class
  convention: before fields it talks about

added to pre-/post condition methods
post ∧ constructor

Inheritance
Spec from superclass

BRUNNEN

JML expr ↔ Java expr

pure Java expr.

\old (expr.) - from prestate

boolean JML expr. a b

!a , &&, ||

a ==> b

a <==> b

Quantified

(\forall type varName; expr.) [varName2]

\exists

Range Predicates

(\forall type varName; $expr_1$; $expr_2$)

\exists   sugar for   ==> - bei ∀

&& - bei ∃

\created (object) - true on created obj

\result

Dynamic Logic – Java in FOL

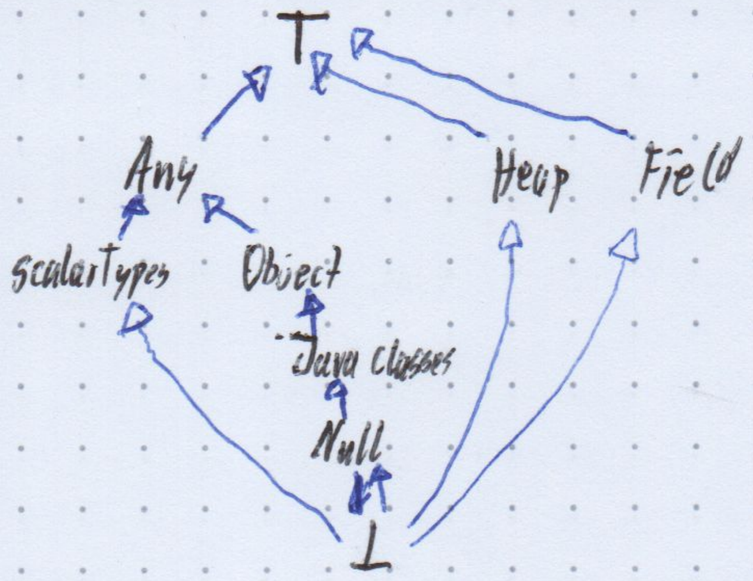
Type Hierarchy $\mathcal{T} = (T_{Sym}, \sqsubseteq)$

T_{Sym} – set of type symbols

$\sqsubseteq \subseteq T_{Sym} \times T_{Sym}$ subtype relation

$\bot$ empty type – no therms of $\bot$

$\top$ universal – untyped FOL



FO/Logical Vars ↪ Program Vars ProgVSym

rigid — non-rigid/flexible

in quantifiers declared — state dependent

not in programms — declared in code

not quantifiable

DL Signature $\Sigma = (P_{Sym}, F_{Sym}, V_{Sym}, ProgV_{Sym})$ $F_{Sym} \cap ProgV_{Sym} = \emptyset$

Predicate, Function, flexible

KeY Input file

```
\sorts {
\functions {
\predicates {
\programVariables { // non-rigid, global
\problems { // formula to verify
```

JavaSource "path";

Modalities

exceptional/abrupt ↕ normally

diamond box

$\langle p \rangle \phi$ – total correctness — p terminates, ϕ holds in final state

$[p]$ partial

Semantik

state $s \in (D, \delta, I)$

for D, δ : S = states with

D, δ – constant Domain assumption

I differs only in flexible vars

Kripke structure / Labeled Transition System

$K = (S, \rho)$

ρ: Program $\to (S \to S)$ – defined by syntactic induction

$\rho(p)(s_1) = s_2 \Leftrightarrow$ p exec in s_1 terminates normally in s_2

Formula validity (for K initial programm var values)

$(K, s, \beta) \models \langle p \rangle \phi \Leftrightarrow \rho(p)(s)$ defined $\wedge (K, \rho(p)(s), \beta) \models \phi$

$[\,] \Rightarrow$

$\models \phi \Leftrightarrow val_{s,\beta}(\phi) = T$

valid $\Leftrightarrow \forall K \forall s \in S \forall \beta : (K, s, \beta) \models \phi$

Form

BRUNNEN

$\langle p \rangle \phi \Leftrightarrow \neg [p] \neg \phi$

$\langle p \rangle \phi \Rightarrow [p] \phi$

DL sequenz

$\{\phi_1, \dots, \phi_n\} \Rightarrow \{\psi_1, \dots \psi_m\} \Leftrightarrow (\phi_1 \land \dots \land \phi_n) \rightarrow (\psi_1 \lor \dots \lor \psi_m)$

Symbolic Execution

follow control flow

Rule Schema

$\Gamma \Rightarrow \langle \pi \ stmt; \ \omega \rangle \phi, \Delta$ (also with [ ])

- ω – postfix
- stmt – active stmt – first executeable
- π – inactive prefix

if $\dfrac{\Gamma, b \doteq true \Rightarrow \langle \pi \ p \ \omega \rangle \phi, \Delta \qquad \Gamma, b \doteq false \Rightarrow \langle \pi \ q \ \omega \rangle \phi \Delta}{\Gamma \Rightarrow \langle \pi \ if(b) \{p\} \ else \{q\} \ \omega \rangle \phi, \Delta}$

Symbolic updates +simplify, combine effects

$\{v := t\}(t' | \phi)$

- v – $\in$ ProgVSym
- t – $\in$ $Term_T$

$p(\{v := t\})(s) = s'$ mit $I_{s'}(v) = val_{s,\beta}(t)$, Rest gleich

Program var $\{x := t\}y \rightsquigarrow y \quad x \neq y$

$x \rightsquigarrow t$

$f(t_1, \dots t_n) \rightsquigarrow f(\{x := t\}t_1, \dots)$

...

no rewrite rule for $\{x := t\}(\langle p \rangle \phi)$

$\forall i_0; \{i := i_0\} \langle \_ i \_ \rangle \phi$

quantify over ProgramVars

Parallel Update

$\{v_1 := r_1 \ || \dots || \ v_n := r_n\}$ @conflict last wins

$\{v_1 := r_1\}\{v_2 := r_2\} = \{v_1 := r_1 \ || \ v_2 := r_2\}$

Heap in FOL

Heap store(Heap, Object, Field, Any)
(Any – value)

Any select(")
(Any – often needs to be cast z.B. int :: select(...))

u.f@heap

Pretty Printing in KeY

- u.f@heap = select
- heap[o.f := t] = store

$\Rightarrow$ ClassName $\in$ TSym

$D^{ClassName}$

unique constant field $\in$ FSym of type Field

ClassName :: $field

assign $\dfrac{o \doteq null, \Gamma \Rightarrow \langle throw \ new \ NullPointerException(); \ \omega \rangle \phi, \Delta \qquad o \,!\!= null, \Gamma \Rightarrow \{heap := store(heap, o, f, t)\} \langle \pi \ \omega \rangle \phi, \Delta}{\Gamma \Rightarrow \langle \pi \ o.f = t; \ \omega \rangle \phi, \Delta}$

null.f defined but unknown

$null.f \doteq null.f$

Aliasing

Static fields

- final static – Compiletime constant $\Rightarrow$ FOL constant $C.f : A$
- else $C.f$: Field store(heap, null, C.f, v)

this : Object, != null
unique program var
often called self

Arrays
flexible function arr : int $\rightarrow$ Field
store(heap, a, arr(i), v)

a.length

Object Creation $\phi \in$ rigid FOL formula

$\vdash \forall x; \phi \rightarrow [p](\forall x; \phi)$

implizit created : Boolean Field

Decomposition of complex terms e.g. expressions with side effects

Principles

Local code Transformation

e.g. $$\frac{\ldots\ x = t;\ x = y; \ldots}{\ldots\ x = y = t; \ldots}$$

fresh var for intermediate res

e.g. $$\frac{\ldots \text{boolean } v0;\ v0 = b;\ \text{if}(v0)\ p; \ldots}{\ldots\ \text{if}(b)\ p; \ldots}$$

Abrupt termination

try{ - inactive Prefix

$$\frac{\Rightarrow \langle \pi\ \text{if}(e \text{ instanceof } T)\{ \text{try}\{x=e;q\} \text{finally}\{r\}\} \text{else}\{r; \text{throw } e;\}\ \omega\rangle \phi}{\Rightarrow \langle \pi\ \text{try}\{\text{throw } e; p\} \text{catch}(T\ x)\{q\}\ \text{finally}\{r\}\ \omega\rangle \phi}$$

Method call

- source needed
  ! commercial API's
  system-level "
- invoked multiple times
  gets big fast
- no unbound recursion

fresh vars for param p#i

Lookup possible impl. class, split proof on ~~dynamic dispatch~~

Polymorphism
dynamic dispatch
Scoping
visibility

Inlining

(rename param)

methodBodyExpand $$\frac{\Gamma \Rightarrow \langle \pi\ \text{method-frame}(\text{source}=C, \text{this}=c):\{\text{body}\}\ \omega\rangle \phi, \Delta}{\Gamma \Rightarrow \langle \pi\ c.m(p\#1,\ldots)@C;\ \omega\rangle \phi, \Delta}$$

Object initialization

chain to new Object

⇒ methods
<createObject>()
<init>()

⇒ (use) Method contracts

$\Gamma \Rightarrow \mathcal{U}(F(preNormal) \vee F(preExc)), \Delta$ (precondition)

" $\mathcal{V}_{mod_{normal}}(\{exc \doteq null \wedge \phi_{post}) \rightarrow \langle \pi\ p\ \omega\rangle \phi), \Delta$ (normal)

$\quad_{exc}$ $\neq$ $\langle \pi\ \text{throw } exc; p\ \omega\rangle$ (exceptional)

$$\overline{\Gamma \Rightarrow \mathcal{U}\langle \pi\ result = m(args, \ldots);p\ \omega\rangle \phi, \Delta}$$

Anonymising Heap of @assignable modified

$\mathcal{V}_{modified} = \{ l_1 = c_1 \| \ldots \}$
- $l \in$ mod., c fresh constant
- $ls \subseteq$ Object × Fields

anon(heap, LocationSet, freshHeap)
= heap like heap if of $\in ls$
like freshHeap sonst

~~requires~~ Proving Method Contracts

$F(preNormal) \rightarrow \langle$ exc = null; try{o.m();} catch (Exception e) {exc = e;} $\rangle$ (exc $\doteq$ null $\wedge$ F(postNormal) $\wedge$ assignable (correct))

exceptional ..

unclosed proofs

Reasons
- buggy/incomplet spec
  program
- max #proof steps
- to weak automatic proof search

Understand/Fix
1. follow flow to open goal
2. Whats unproveable?

Loop invariants for loops with many iterations (hard with unwind)

at start
preserved by loop guard+body ⇒ holds afterwards
terminates

↳ by induction
Hypothesis, Base, Stepcase

How to find Inv.
desired postcondition?
Strengthening
Generalization partial results?

preserve context unmodified by loop

! default assignable \everything

$$\frac{\Gamma \Rightarrow \mathcal{U}(Inv \wedge b \doteq TRUE \to [p]Inv),\Delta \quad \Gamma \Rightarrow FALSE \to [\pi\, w]\emptyset,\Delta}{\Gamma \Rightarrow \mathcal{U}[\pi\, \text{while}(b)\, p\, w]\emptyset,\Delta}$$

initially valid
preserved
use case

```
/*@ loop_invariant
  Ø;
  assignable ...;
  diverges true;/
  decreasing v;
```

{ v ≥ 0 ; v strictly dec. } by loop body

e.g. a.length - i

Termination?

Well-Founded Order over decreasing term/variant