Model Checking for Industrial Applications

Bernd-Holger Schlingloff
Bremen Institute of Safe Systems
University of Bremen, Germany
hs@tzi.de

Tutorial @ ICFEM '98, Dec. 9^{th} 1998, Brisbane, Australia

Bremen Institute of Safe Systems

- One of five departments in the TZI (Technologie-Zentrum Informatik)
- Technology-transfer institute within Bremen university
- 30 Scientists, 5 Professors, 1.2 Mio DM grant money
- Application of formal methods in industrial contexts
- Cooperation with aerospace, railway, telecommunication companies
- Specification, Simulation, Formal Testing, Modelling, *Verification*

Contents

1. Formal methods in software design

Scope of this tutorial

- Quality assurance in software development
- Formal design methods
- Safety-critical systems
- Reactive programs
- Control-oriented programming paradigm
- Concurrent and distributed software
- Embedded and real-time applications

Examples: Protocols, circuits, controllers, finite state algorithms, ...
Not: Database systems, multimedia, graphical user interfaces, ...

Software quality assurance

Why is it important?

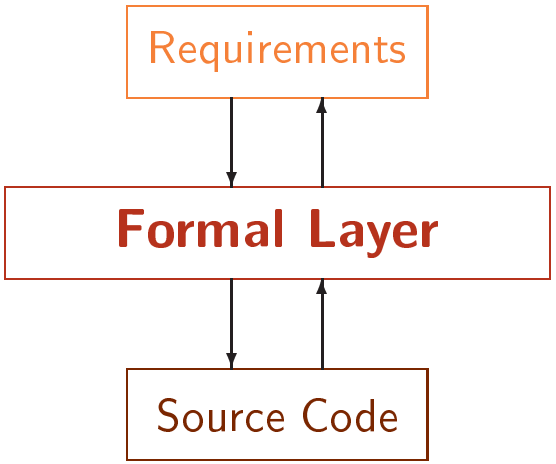
- Correctness of software is in many cases crucial
- High probability of errors in complex systems
- Cost of error correction increases during project lifetime
- Errors often in unpredicted situations

Why are conventional methods insufficient?

- Reviews and code inspections do not capture dynamic behaviour
- Software metrics and coding standards do not help much
- Static and dynamic analysis cannot find intricate errors
- esting often explores only standard paths

Formal Methods

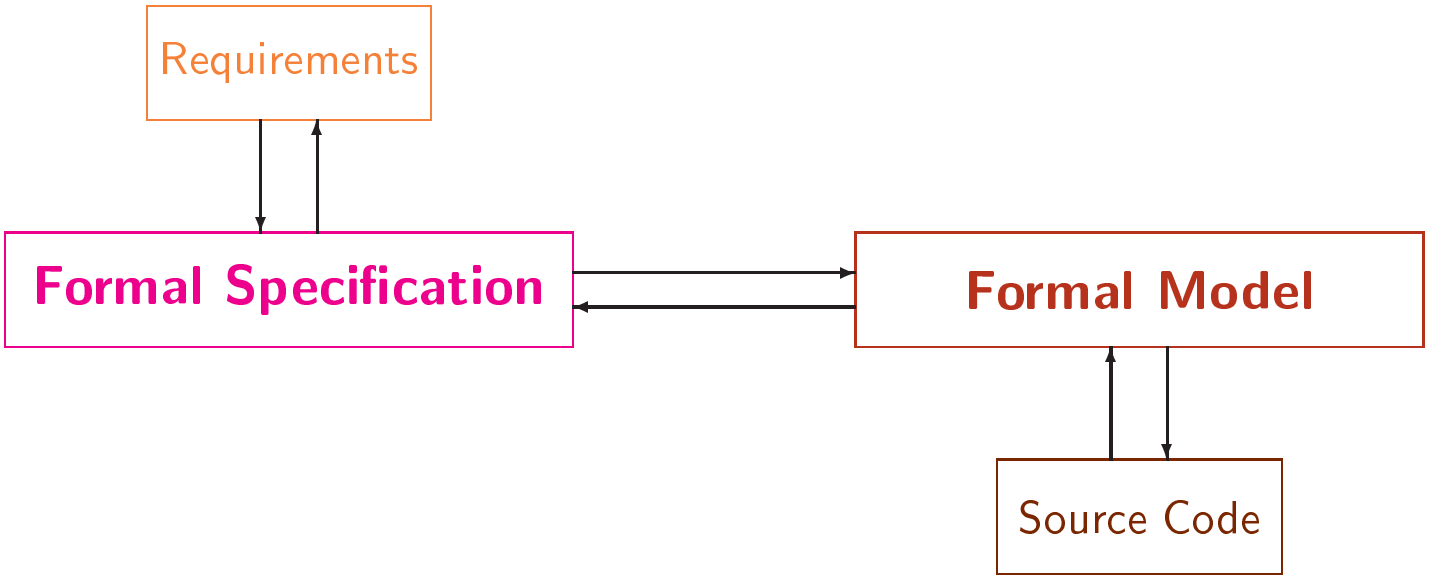
Due to advances in theory and available computing power, formal methods have come of age!



Possibility of

- Prototyping, synthesis and transformation
- Formal simulation, testing and debugging
- Verification, program proving, model checking

Verification



Specification languages: MSC, HOL, Z, Temporal Logic, ...
Modelling languages: SDL, StateCharts, CSP, Transition Systems, ...

Interactive vs. Automatic Verification

Interactive methods
general
hard to use

Automatic verification
limited
easy to use

Interactive system:

- checks proof steps
- lists assumptions and subgoals,
- applies indicated rules
- suggests rules

– usually expert knowledge required
– not accepted by system engineers

Automatic system:

model + property
$\Rightarrow$ proof or counter-model

Finiteness requirement:

- Physical machines finite
- Finite control structure
- Infinite domains unnecessary

Model Checking in Quality Assurance

Exhaustive search of the model state space for the given specification.

Some advantages:

- Completely automatic, usually fast
- Counterexamples for debugging
- Various properties of the same model can be checked

Some disadvantages:

- Finiteness requirement, data abstraction
- State explosion problem
- Model building and updating from code

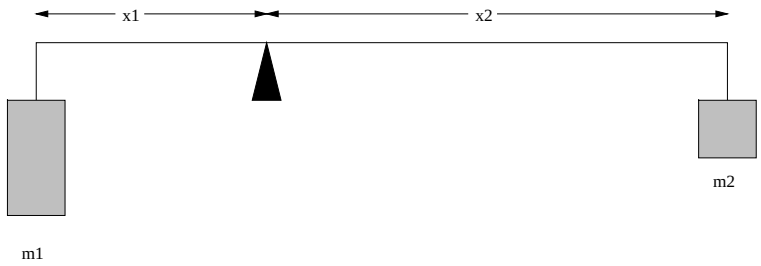
Developed in the 80's, advances and case studies in the 90's
Standard method in the next decade?

The ruler and the loads

(Example by J. Moore)

For several millenia, mankind has been using mathematical models to describe and predict the behaviour of physical systems. Theorems are proved about formal models and abstract properties, not about real systems; we can never know to which extent our models and properties reflect reality.

Archimedes: Commensurable magnitudes are in equilibrium at distances reciprocally to the weights.



That is, $m_1 x_1 = m_2 x_2$.

But, the mathematics necessary to describe even simple systems may be complicated:

$$m_1 x_1 + \int_0^{x_1} \rho x dx = m_2 x_2 + \int_0^{x_2} \rho x dx.$$

Lessons:

Theorems are not proved about the physical systems, theorems are proved about the mathematical models. Models are written down in some notation and must be corroborated for accuracy.

No amount of mathematics or calculation can guarantee that a physical system will not fail in unexpected (i.e., unmodeled) ways.

To reject formal models because they cannot offer such guarantees would be to ignore the most powerful tool mankind has devised for dealing with complex systems.

2. Specification and modelling languages

Propositional logic

Assume a basic propositions $\mathcal{P}$ (e.g., "`sun_is_shining`" or "`stackptr=nil`"), which can be either true or false.

Syntax of propositional logic **PL**:

$$\mathbf{PL} \quad ::= \quad \mathcal{P} \quad | \quad \bot \quad | \quad (\mathbf{PL} \to \mathbf{PL})$$

That is,

- Every $\mathtt{p} \in \mathcal{P}$ is a well-formed formula
- $\bot$ is a well-formed formula ("falsum")
- $(\varphi \to \psi)$ is well-formed, if φ and ψ are.

Other connectives are defined as usual: $\neg\varphi$, $\top$, $(\varphi \vee \psi)$, $(\varphi \wedge \psi)$, $(\varphi \leftrightarrow \psi)$

Semantics of PL:

Interpretation $\mathcal{I} : \mathcal{P} \mapsto \{\mathbf{true}, \mathbf{false}\}$
Wittgenstein (Tractatus logico-philosophicus): "The world is all which is the case"

Let $\mathcal{I} \models \varphi$ iff $\mathcal{I}(\varphi) = \mathbf{true}$.

- $\mathcal{I} \not\models \bot$, and
- $\mathcal{I} \models (\varphi \to \psi)$ iff $\mathcal{I} \models \varphi$ implies $\mathcal{I} \models \psi$.

This simple formalism can model a wealth of systems.
E.g., constraint satisfaction problems, combinational circuits, dependency diagrams, decision tables, finite sets, finite relations and functions, all sorts of programs, ...

Propositional logic is not well-suited to formalise statements about time.

Temporal logic

provides an additional operator $(\varphi\ \mathbf{U}^{+}\ \psi)$ meaning "φ holds until ψ holds"

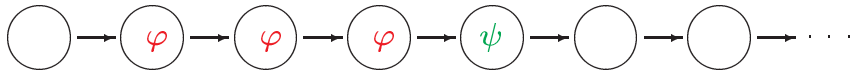
$$\mathbf{TL} \quad ::= \quad \mathcal{P} \ \mid\ \bot \ \mid\ (\mathbf{TL} \to \mathbf{TL}) \ \mid\ (\mathbf{TL}\ \mathbf{U}^{+}\ \mathbf{TL})$$

Kripke model $\mathcal{M} \triangleq (U, \longrightarrow, \mathcal{I}, w_0)$
— Universe U of points in time
— Accessibility relation between points: $\longrightarrow \subseteq U \times U$
— Interpretation $\mathcal{I} : U \times \mathcal{P} \mapsto \{\mathbf{true}, \mathbf{false}\}$
— Initial point $w_0 \in U$.

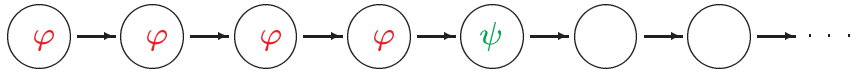
"$<$" denotes the transitive closure of "$\longrightarrow$"
Notation: $w \models \varphi$ iff $(U, \longrightarrow, \mathcal{I}, w) \models \varphi$

Definition of semantics:

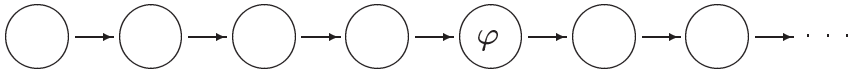
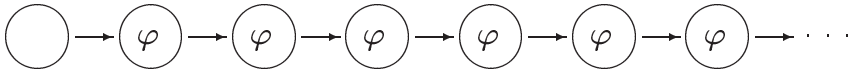
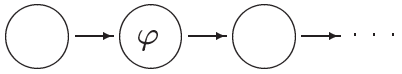
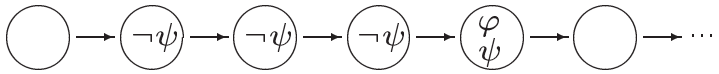
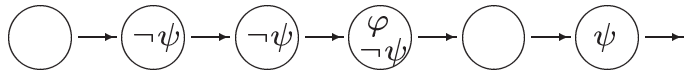
- $w \not\models \bot$, and $w \models (\varphi \to \psi)$ iff $w \models \varphi$ implies $w \models \psi$;
- $w \models \mathtt{p}$ iff $w \in \mathcal{I}(w, \mathtt{p})$;
- $w \models (\varphi \ \mathbf{U}^{+} \ \psi)$ iff $v \models \psi$ for some $v > w$, and $u \models \varphi$ for $w < u < v$.



- $w \models (\varphi \ \mathbf{U}^{*} \ \psi)$ iff $v \models \psi$ for some $v \geq w$, and $u \models \varphi$ for $w \leq u < v$.



Other connectives

- *sometime*: $\mathbf{F}^+ \varphi \stackrel{\Delta}{=} (\top \mathbf{U}^+ \varphi)$ 
- *always*: $\mathbf{G}^+ \varphi \stackrel{\Delta}{=} \neg \mathbf{F}^+ \neg\varphi$ 
- *next-time*: $\mathbf{X} \varphi \stackrel{\Delta}{=} (\bot \mathbf{U}^+ \varphi)$ 
- *atnext*: $(\varphi \mathbf{A}^+ \psi) \stackrel{\Delta}{=} (\neg\psi \mathbf{U}^+ (\varphi \wedge \psi))$ 
- *before*: $(\varphi \mathbf{B}^+ \psi) \stackrel{\Delta}{=} (\neg\psi \mathbf{U}^+ (\varphi \wedge \neg\psi))$ 

Linear and branching time logics

Executions of a program =

- set of execution sequences, or
- single execution tree

LTL (linear temporal logic) is interpreted on **sequences**.
(Syntactically, there is no difference between **TL** and **LTL**!)

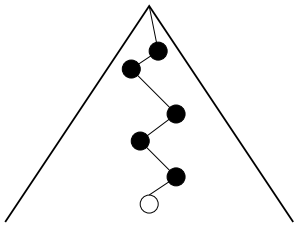
Syntax of **CTL** (computation tree logic):

$$\mathbf{CTL} ::= \mathcal{P} \mid \bot \mid (\mathbf{CTL} \rightarrow \mathbf{CTL}) \mid \mathbf{E}(\mathbf{CTL}\ \mathrm{U}^{+}\ \mathbf{CTL}) \mid \mathbf{A}(\mathbf{CTL}\ \mathrm{U}^{+}\ \mathbf{CTL})$$

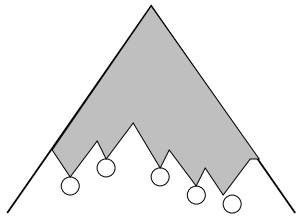
Semantics of CTL:

CTL is interpreted on **trees**, where $<$ is the usual tree-order.

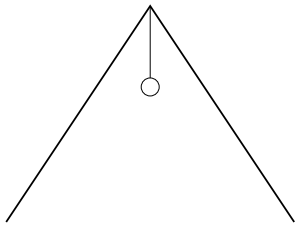
- $w_0 \models \mathbf{E}(\psi\ \mathbf{U}^+\ \varphi)$ iff $\exists\ w_1 > w_0,\ w_1 \models \varphi,\ \forall\ w_0 < w_2 < w_1,\ w_2 \models \psi$
- $w_0 \models \mathbf{A}(\psi\ \mathbf{U}^+\ \varphi)$ iff *for all paths* p from w_0,
  $\exists\ w_1 > w_0$ *on path* p s.t. $w_1 \models \varphi$, and $\forall\ w_0 < w_2 < w_1,\ w_2 \models \psi$



some path



all paths



some successor

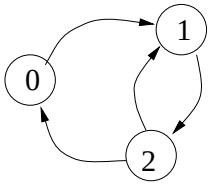
Expressivity LTL versus CTL

Models are different!

To compare, we use general Kripke models $(U, \longrightarrow, \mathcal{I}, w_0)$

— **LTL**-validity for all *generated sequences*

— **CTL**-validity for the *generated tree*

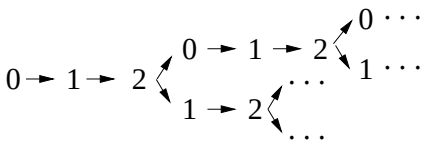


$0 \to 1 \to 2 \to 0 \to 1 \to 2 \to 0 \to \cdots$

$0 \to 1 \to 2 \to 0 \to 1 \to 2 \to 1 \to \cdots$

$0 \to 1 \to 2 \to 1 \to 2 \to 0 \to \cdots$

$0 \to 1 \to 2 \to 1 \to 2 \to 1 \to \cdots$



$\mathbf{F}^+\mathbf{G}^+\,\mathrm{p}$ not expressible in **CTL**

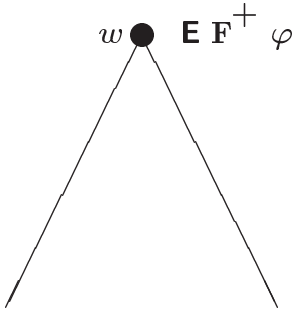
$\mathbf{E}\,\mathbf{F}^+\,\mathbf{A}\,\mathbf{G}^+\,\mathrm{p}$ not expressible in **LTL**

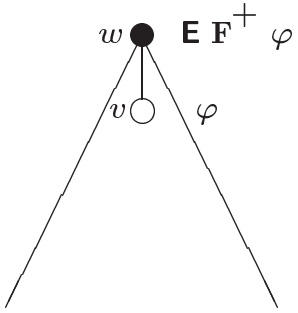
3. Model checking algorithms

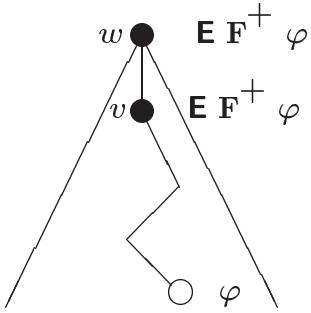
Model checking problem:
Given finite Kripke model $\mathcal{M} = (U, \longrightarrow, \mathcal{I}, w_0)$ and formula φ, check if

$$\mathcal{M} \models \varphi$$

For **CTL**: Recursive descent on subformulas.
$w \models \mathbf{E}\,\mathbf{F}^+\,\varphi$ iff $\exists w \longrightarrow v$ s.t. $v \models \varphi$ or $\exists w \longrightarrow v$ s.t. $v \models \mathbf{E}\,\mathbf{F}^+\,\varphi$

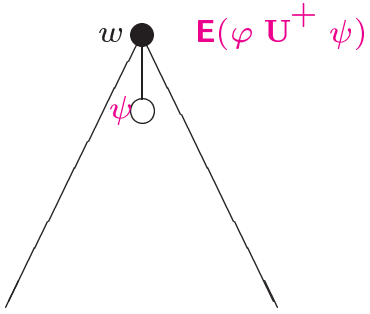


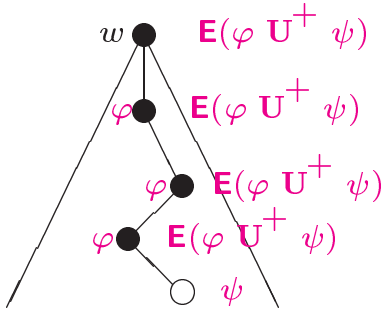


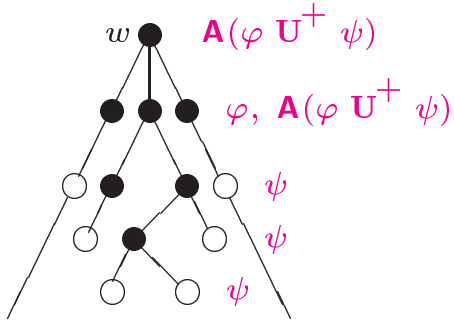


Similarly,

- $w \models \mathbf{E}(\varphi\ \mathbf{U}^+\ \psi)$ iff
  for some $w \longrightarrow v$ it holds that $v \models \psi$ or $v \models \varphi$ and $v \models \mathbf{E}(\varphi\ \mathbf{U}^+\ \psi)$
- $w \models \mathbf{A}(\varphi\ \mathbf{U}^+\ \psi)$ iff
  for all $w \longrightarrow v$ it holds that $v \models \psi$ or $v \models \varphi$ and $v \models \mathbf{A}(\varphi\ \mathbf{U}^+\ \psi)$





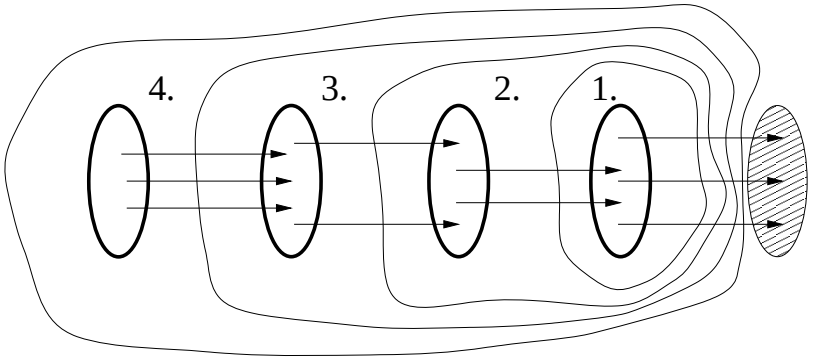


Let $\varphi^{\mathcal{M}} = \{w \mid w \models \varphi\}$.

$(\mathbf{E}\ \mathbf{F}^{+}\ \varphi)^{\mathcal{M}}$ is the set of points from which some point in $\varphi^{\mathcal{M}}$ is reachable.
How to determine $(\mathbf{E}\ \mathbf{F}^{+}\ \varphi)^{\mathcal{M}}$ from $\varphi^{\mathcal{M}}$? (*Inverse reachability problem*)

Backward iteration marks all points in $(\mathbf{E}\ \mathbf{F}^{+}\ \varphi)^{\mathcal{M}}$:

- Initially mark all points for which some direct successor is in $\varphi^{\mathcal{M}}$.
- Repeatedly add all points which have some marked successor.



procedure eval (Formula φ, Model $\mathcal{M}$): Pointset $=$

case φ **of**

$\mathtt{p}$: **return** $\mathcal{I}(\mathtt{p})$;

$\bot$: **return** $\{\}$;

$(\varphi \rightarrow \psi)$: **return** $U \setminus \mathtt{eval}(\varphi, \mathcal{M}) \cup \mathtt{eval}(\psi, \mathcal{M})$;

$\mathbf{E}(\varphi\ \mathbf{U}^{+}\ \psi)$: $E1 := \mathtt{eval}(\psi, \mathcal{M})$; $E2 := \mathtt{eval}(\varphi, \mathcal{M})$; $E := \{\}$;

repeat until stabilization

$E := E \cup \{w \mid (\mathtt{succ}(w) \cap (E1 \cup E2 \cap E)) \neq \{\}\}$;

return E;

$\mathbf{A}(\varphi\ \mathbf{U}^{+}\ \psi)$: $E1 := \mathtt{eval}(\psi, \mathcal{M})$; $E2 := \mathtt{eval}(\varphi, \mathcal{M})$; $E := \{\}$;

repeat until stabilization

$E := E \cup \{w \mid \mathtt{succ}(w) \subseteq E1 \cup E2 \cap E\}$;

return E;

Complexity: linear in formula, quadratic in model $(|sf(\varphi)| \cdot |U| \cdot | \longrightarrow |)$

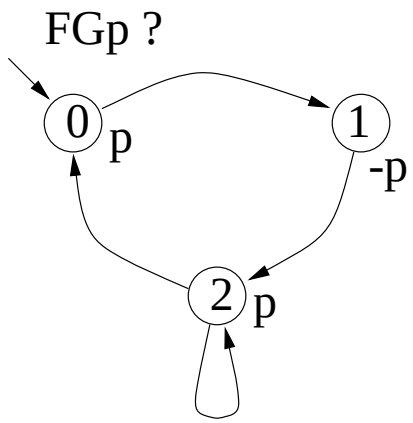
Improvement: quadratic in $|U|$, i.e., linear in $\mathcal{M}$, is possible

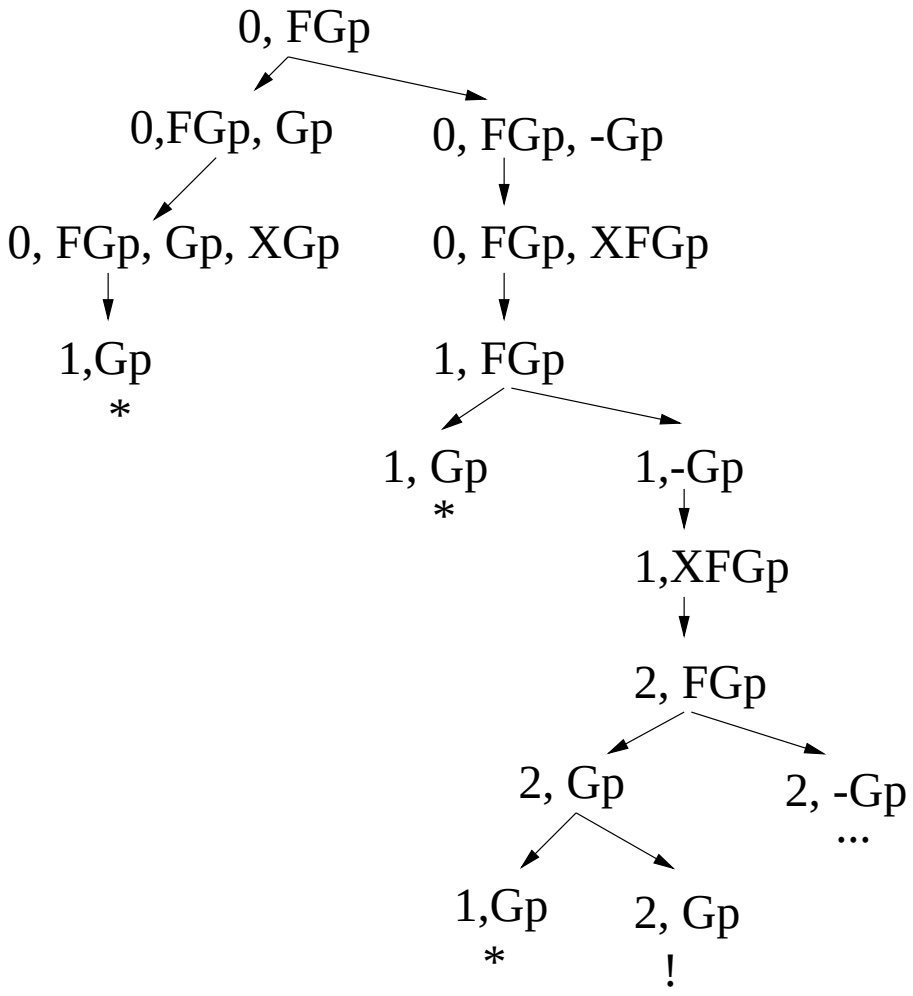
LTL model checking

Given model $\mathcal{M}$ and formula φ, check whether $\sigma \models \varphi$
for some sequence σ which can be extracted from $\mathcal{M}$.
Somewhat harder question than for **CTL**!
In any point w, an **LTL**-formula φ can be **true** for one sequence and **false** for another one.

Idea: Depth-first-search for an appropriate path σ in the model.
"On-the-fly" checking whether the current path satisfies φ

Consider all subformulas of the given formula!
Formally, an *atom* consists of a point and a truth assignment.

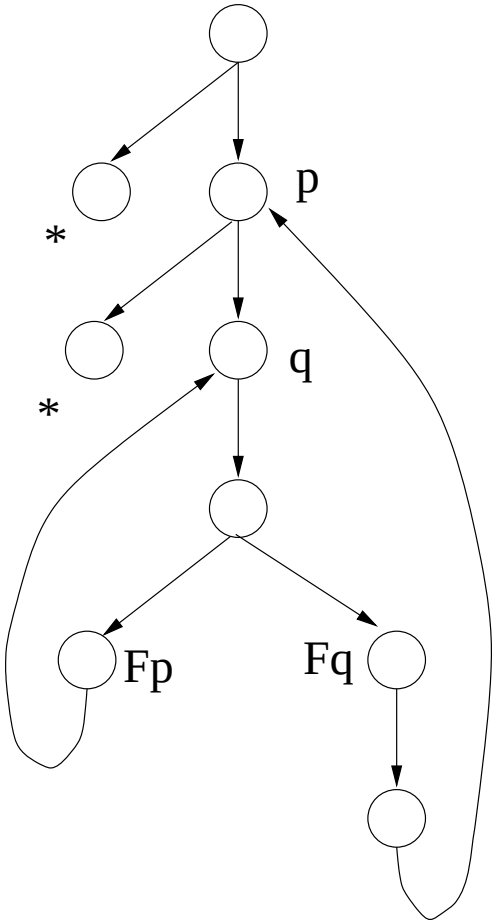

FGp ?
0
p
1
-p
2
p


0, FGp
0,FGp, Gp
0, FGp, -Gp
0, FGp, Gp, XGp
0, FGp, XFGp
1,Gp
*
1, FGp
1, Gp
*
1,-Gp
1,XFGp
2, FGp
2, Gp
2, -Gp
...
1,Gp
*
2, Gp
!

Whenever a cycle is detected, we check if all $\mathbf{F}^+$-subformulas are satisfied.

$\mathbf{F}^+$ φ is satisfied if there is some "reachable" atom containing φ

Tarjans algorithm enumerates SCC's during DFS.



```
procedure depth_first_search (Atom α) =
    if (table [α] = UNDEFINED) then    /* α is a new atom */
        Nat dfnumber := depth_first_count;    /* save current count */
        depth_first_count := depth_first_count+1;
        table[α] := dfnumber;    /* initialize with current depth */
        push(stack, α);    Atomset succ := children(α);
        for all (β ∈ succ) do
            depth_first_search(β);
            table[α] := min(table[α], table[β]);    /* β above α? */
        if (table[α] = dfnumber) then    /* α is the root of an SCC */
            Formulaset required := {},  fulfilled := {};
            repeat
                β := pop(stack);
                table[β] := MAXNAT;
                required := required ∪ {ψ1 | (ψ2 U+ ψ1) ∈ β};
                fulfilled := fulfilled ∪ {ψ | ψ ∈ β}
            until (α = β);    /* all elements of SCC are popped */
            if required ⊆ fulfilled    /* SCC is self-fulfilling */
            then print("φ satisfiable in M"); exit;
```

Refinement model checking

Transition system: (Σ, S, Δ, S_0), where

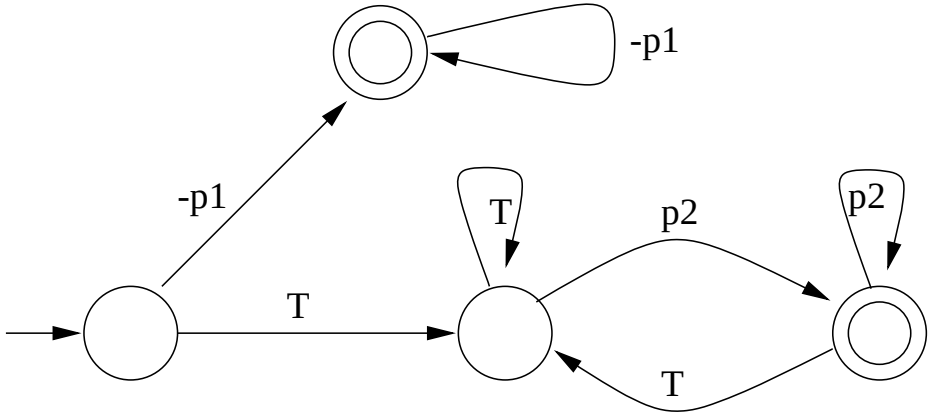
- Σ is a nonempty finite *alphabet*,
- S is a nonempty finite set of *states*,
- $\Delta \subseteq S \times \Sigma \times S$ is the *transition relation*, and
- $S_0 \subseteq S$ is the set of *initial states*.

For every transition system there exists a unique Kripke model.

Büchi automaton: $(\Sigma, S, \Delta, S_0, S_{acc}, S_{rec})$, where

- $S_{acc} \subseteq S$ is the set of accepting states (for finite words), and
- $S_{rec} \subseteq S$ is the set of recurring states (for infinite words).

Büchi automata are used to specify set of ω-words



Language: $(\neg \mathtt{p1})^{\omega} + (\top^*; \mathtt{p2})^{\omega}$. **LTL**-formulation: $(\mathbf{G}^* \neg \mathtt{p1} \vee \mathbf{G}^* \mathbf{F}^+ \mathtt{p2})$.

Automata specifications	**Logical specifications**
graphical	textual
operational	descriptive

For any model there is a unique transition system (automaton).
For any **LTL** formula there is a corresponding Büchi automaton.
Thus, model checking can be reduced to automata inclusion:

$$\mathcal{M} \models \varphi \quad \text{iff} \quad L(\mathcal{M}_{\mathcal{A}}) \subseteq L(\mathcal{M}_{\varphi})$$

- build automaton which corresponds to the (negated) formula
- construct product of model and formula automaton
- check whether the generated language is nonempty

"On-the-fly" (pipelined) execution gives similar DFS algorithm as above

4. Binary decision diagrams

Size of model representation is critical for model checking.
We need an efficient representation of finite sets and relations.

Example: Consider the domain $D \triangleq \{0..15\}$

symbolic: $S \triangleq \{x \mid x \bmod 2 = 0 \vee x > 12\}$

enumeration: $S = \{0, 2, 4, 6, 8, 10, 12, 13, 14, 15\}$

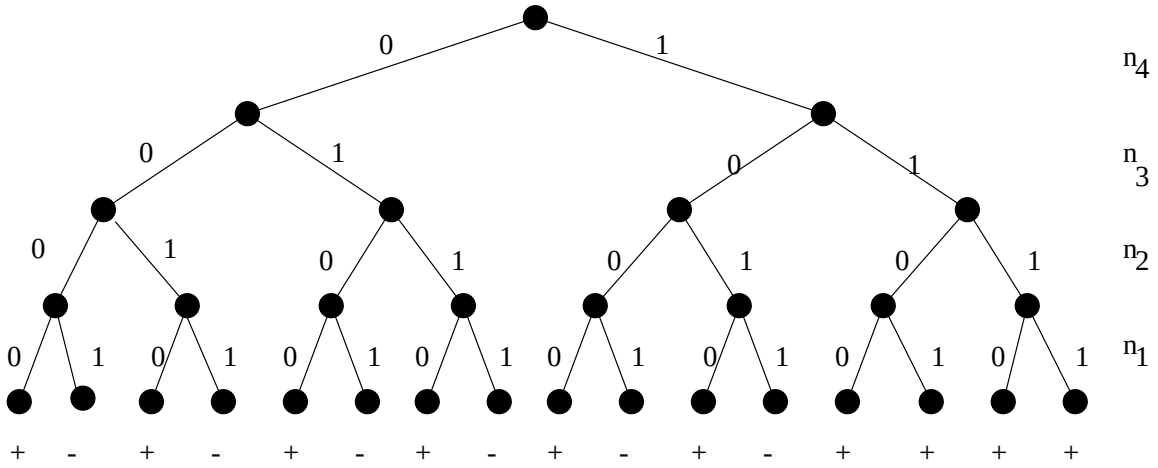
bitstring: $S = (1010101010101111)$

binary: $S = \{0000, 0010, 0100, 0110, 1000, 1010, 1100, 1101, 1110, 1111\}$

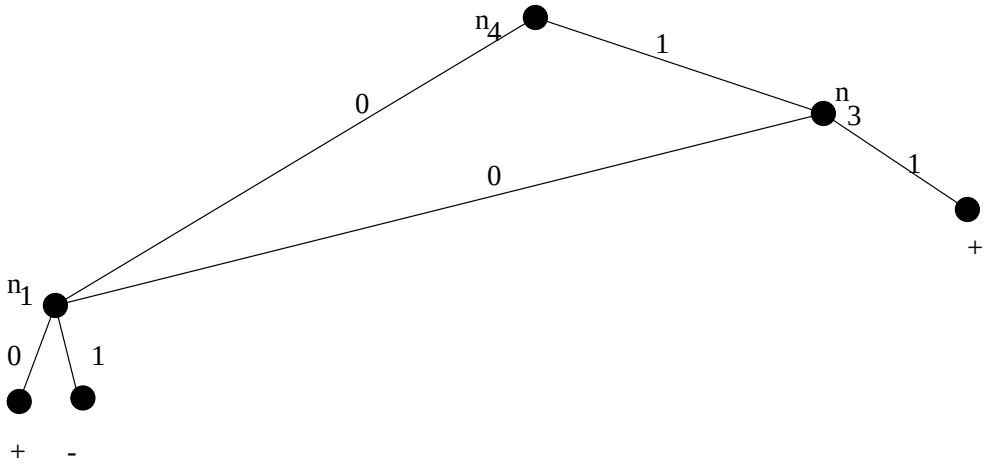
propositional formula: $S = \{n \mid n_1 = 0 \vee n_4 = 1 \wedge n_3 = 1\}$

logical spectrum: $S = \mathsf{Ite}(n_4, \mathsf{Ite}(n_3, \top, \mathsf{Ite}(n_1, \bot, \top)), \mathsf{Ite}(n_1, \bot, \top))$

decision tree:

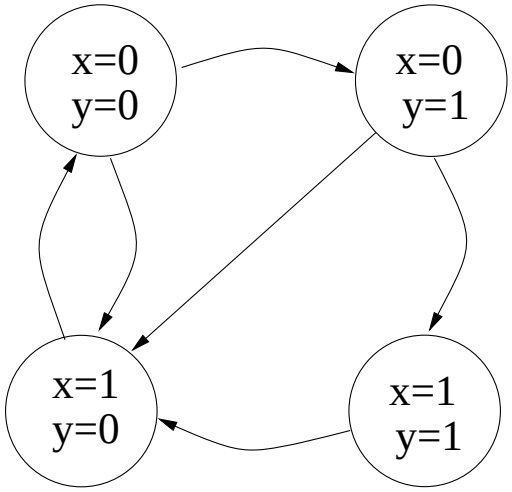


decision diagram:



Size of BDD representation depends on the *structure* rather than the *cardinality* of the represented set
Very large sets of states can be represented ("10^{20} and beyond")
Ordering of state variables important

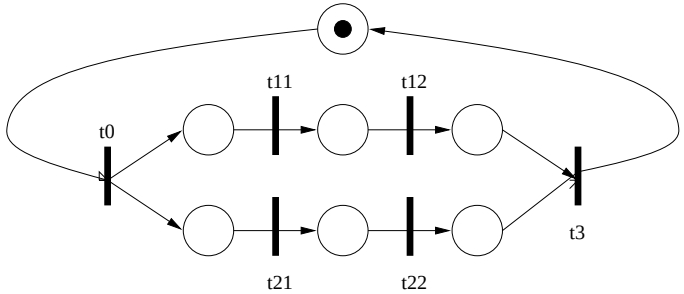
Relations are sets of pairs
Transition relation are defined by state variables and next state variables.



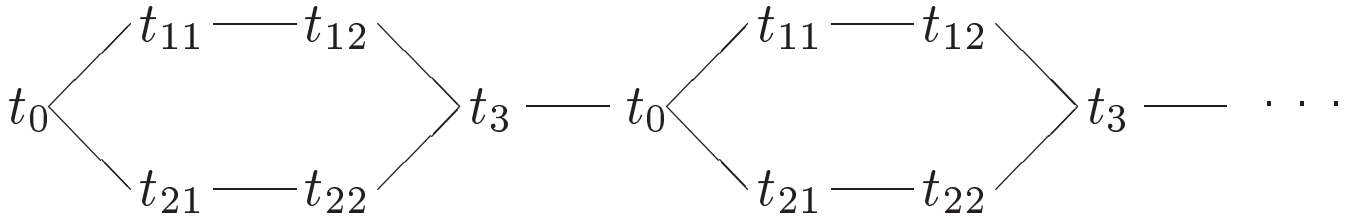
x	y	x'	y'
0	0	0	1
0	1	1	1
1	1	1	0
1	0	0	0
0	0	1	0
0	1	1	0

5. Partial order reductions

State space explosion partially caused by parallelism
usually modelled by interleaving
independent action sequences can have many equivalent interleavings



This system generates the following partial order:



Some of the interleaving sequences

$t_0\ t_{11}\ t_{12}\ t_{21}\ t_{22}\ t_3\ ...$
$t_0\ t_{11}\ t_{21}\ t_{12}\ t_{22}\ t_3\ ...$
$t_0\ t_{11}\ t_{21}\ t_{22}\ t_{12}\ t_3\ ...$
$t_0\ t_{21}\ t_{11}\ t_{22}\ t_{12}\ t_3\ ...$
$t_0\ t_{21}\ t_{11}\ t_{12}\ t_{22}\ t_3\ ...$

Stuttering invariance: Temporal logic without next is insensitive to repetition of identical states.

For stuttering invariant formulas, it is sufficient to consider only one of a class of different interleavings during DFS analysis.

Significant reduction of state space possible!

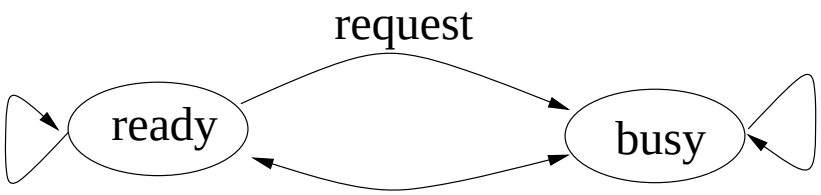
1. Checking sequential circuits with SMV

SMV: **S**ymbolic **M**odel **V**erifier for **CTL**

- Developed since 1987 at CMU by Clarke, McMillan, Berezin et al.
- Symbolic representation of models with BDDs
- Automatic reordering, partitioned transition relations etc.
- Extension for word level model checking available
- Freely available at `www.cs.cmu.edu`
- Emacs interface, command line handling
- Input of model and formula in C-like assignment language

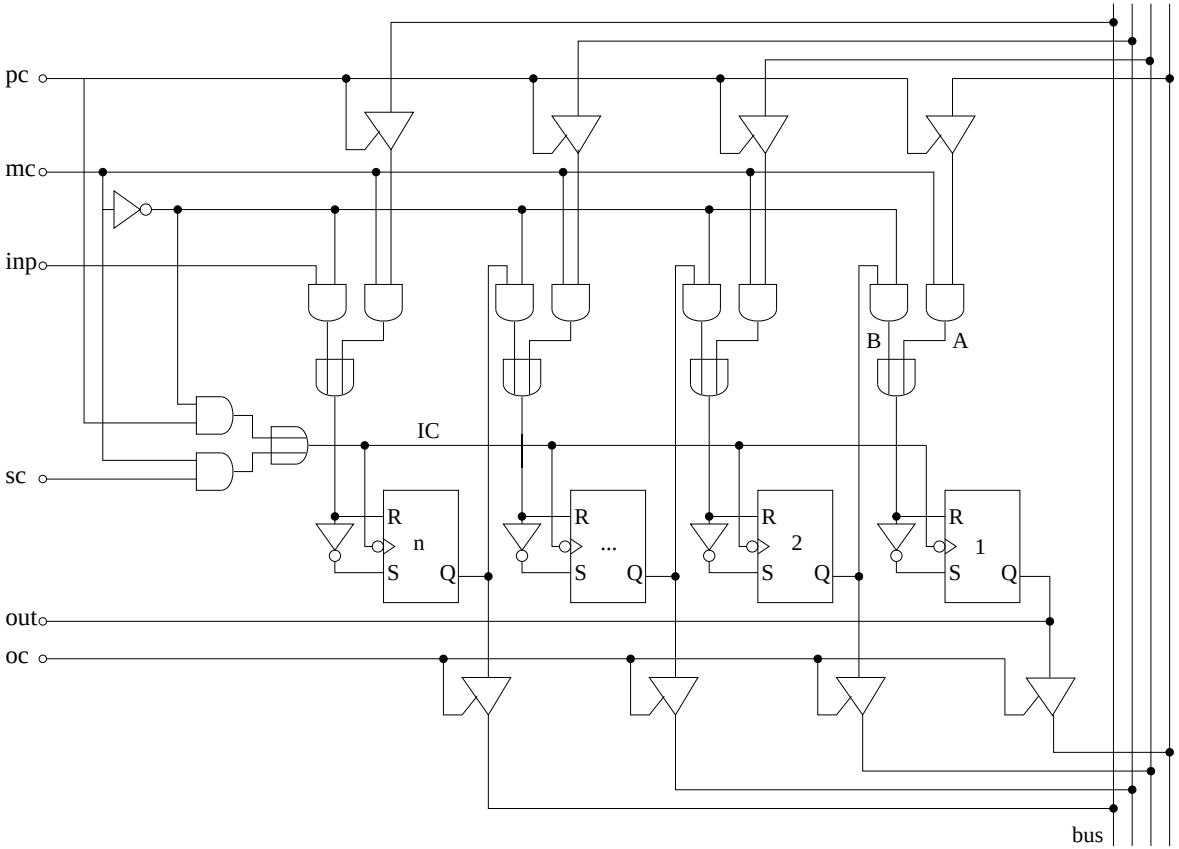
A simple example program

```
MODULE main
VAR request:  boolean;
    state:  {ready, busy};
ASSIGN
  init(state) := ready;
  next(state) := case
    state=ready & request: busy;
    1:  {ready, busy}; esac;

DEFINE is_busy := state = busy;
SPEC AG(request -> AF is_busy)
SPEC AG(request -> AG is_busy)
```




```
-- specification AG (request -> AF is_busy) is true
-- specification AG (is_busy -> AG is_busy) is false
-- as demonstrated by the following execution
-- loop starts here --
state 1.1: is_busy = 0
           request = 0
           state = ready
state 1.2: is_busy = 1
           state = busy
state 1.3: is_busy = 0
           state = ready
resources used:
BDD nodes allocated: 216; Bytes allocated: 917504
BDD nodes representing transition relation: 5 + 1
```

A shift register for data bus interfacing (74x95 TTL family)



- `mc`: mode control
- `pc`: parallel clock
- `sc`: serial clock
- `oc`: output clock
- `inp`: serial input
- `out`: serial output

SR-Latches:

S	R	Q'
0	0	Q
1	0	1
0	1	0
1	1	-

Verification model of shift register

```
MODULE main
VAR Q, bus:  array 1..n of boolean;  -- n SR-latches, n databits
    inp, mc, pc, sc, oc:  boolean;   -- input lines
DEFINE out := Q[1]; ic := ((mc & pc)  |  (!mc & sc));
       A[i] := mc & pc & bus[i]; B[i] := !mc & Q[i + 1];
       R[i] := !(A[i] | B[i]); S[i] := !R[i];
ASSIGN next(Q[i]) := case ic:  case
                               !S[i] & !R[i]:  Q[i];        --hold
                                S[i] & !R[i]:  1;           --set
                               !S[i] &  R[i]:  0;           --reset
                                S[i] &  R[i]:  {0,1}; esac; --undef
                         !ic:  Q[i]; esac; -- unchanged if no input
       next(bus[i]) := case oc:  Q[i];  !oc:  {0, 1}; esac;
FAIRNESS ic FAIRNESS oc
```

Correctness properties:

$$\mathbf{A}\,\mathbf{G}^{*}\,(\mathtt{mc} \wedge \mathtt{pc} \rightarrow (\mathtt{bus[i]} \leftrightarrow \mathbf{A}((\mathtt{oc} \rightarrow \mathbf{A}\,\mathbf{X}\,\mathtt{bus[i]})\,\mathbf{U}^{+}\,\mathtt{ic})))$$

$$\mathbf{A}\,\mathbf{G}^{*}\,(\neg\mathtt{mc} \wedge \mathtt{sc} \rightarrow (\mathtt{Q[i]} \leftrightarrow \mathbf{A}(\mathtt{Q[i\text{-}1]}\,\mathbf{U}^{+}\,\mathtt{ic})))$$

proved for 32 bit bus in less than a second

Similar formulas for verification of more complex properties

Much bigger circuits can be handled if they are "well-behaved":

> there exists an ordering of all wires such that the next state of each wire depends only on wires which are "closely related"

Highly relevant in VLSI and board design

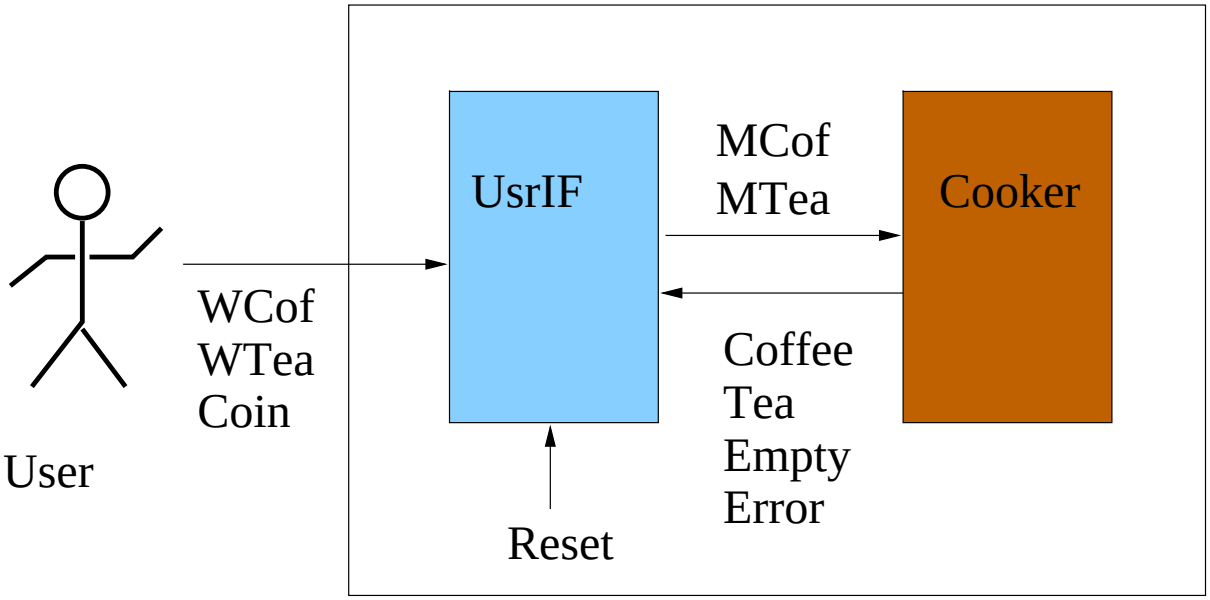
2. Modelling a communication protocol in SVE

SVE: Siemens **S**ystem **V**erification **E**nvironment

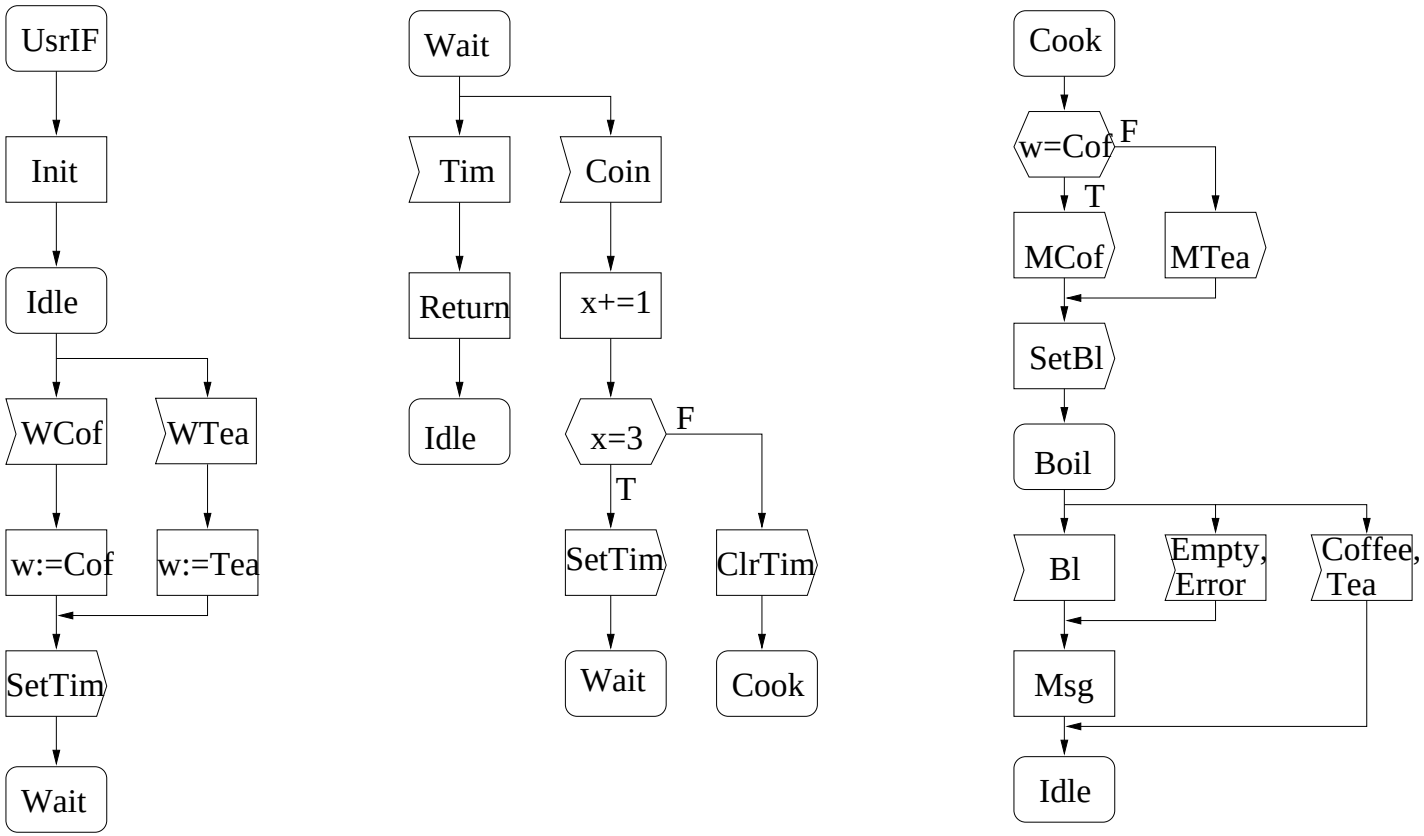
- Commercial system
- Functionality similar to SMV
- Implemented mainly in Prolog
- Prolog-like model description language SVEL
- **CTL** model checker with **LTL** constraints
- Front ends to StateCharts, SDL, SICAT
- Front end specification language SPL

Example: Coffee machine

Communication structure:



SDL-Description:



SPL-Properties:

Definition of temporal intervals:

```
INTERVAL BlRunning IS
   STARTING AT OUTPUT SetBl ENDING AT EVENT Bl
```

Definition of temporal properties:

```
TEMPORAL NeverStartRunningTimer IS
   NOT OCCURS SetBl INSIDE SOME INTERVAL BlRunning
NEVER OUTPUT SetBl INSIDE EACH INTERVAL BlRunning
```

A data dependent property:

```
TEMPORAL NeverWrongCoffee IS
   NEVER OUTPUT MTea INSIDE EACH INTERVAL
         STARTING AT EVENT WCof ENDING AT STATE Idle
```

Assumption to exclude impossible runs:

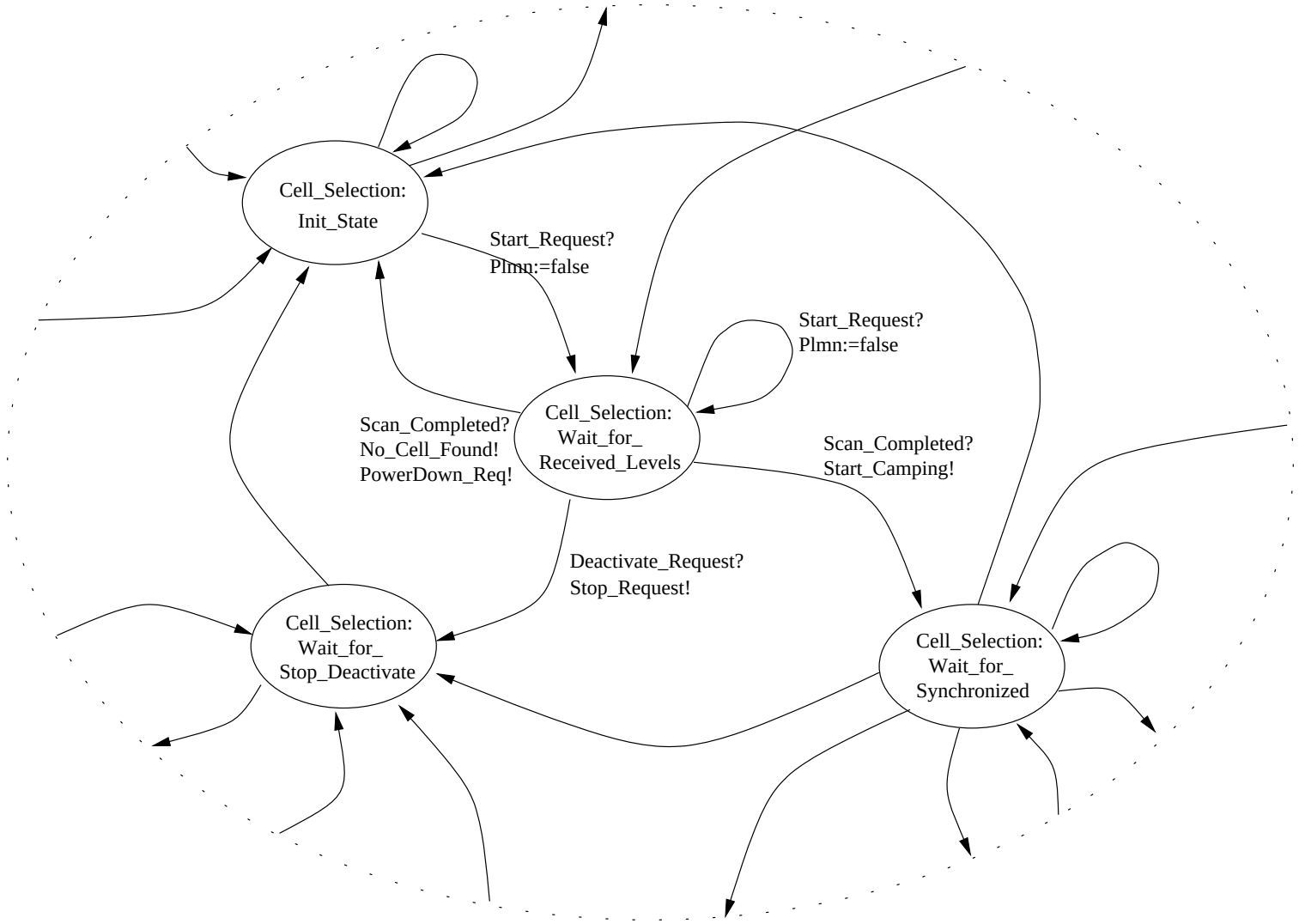
```
ASSUMPTION w IS
   NEVER DECISION (w=Cof IS F) INSIDE EACH INTERVAL
         STARTING AT EVENT WCof ENDING AT STATE Idle
```

Mobile phone operating system (Siemens S6 GSM)

- High number of units sold
- Reuse of basic design
- Five basic processes plus OS kernel
- Priority scheduling of processes
- 50 types of messages
- Priority delivery of messages

Quote from the GSM international standard:

> *"Initially the MS looks for a cell which satisfies the suitability constraints by checking cells in descending order of received signal strength. If a suitable cell is found, the MS camps on it and performs any registration necessary."*


Cell_Selection:
Init_State
Start_Request?
Plmn:=false
Start_Request?
Plmn:=false
Cell_Selection:
Wait_for_
Received_Levels
Scan_Completed?
No_Cell_Found!
PowerDown_Req!
Scan_Completed?
Start_Camping!
Deactivate_Request?
Stop_Request!
Cell_Selection:
Wait_for_
Stop_Deactivate
Cell_Selection:
Wait_for_
Synchronized


Cell_Selection:
Wait_for_
Received_Levels
Start_Request
from
Radio_Resource
Deactivate_Request
from
Radio_Resource
Scan_Completed
from
Power_Handler
PlmnList := false
PlmnList
true
false
Stop_Request
to
Power_Handler
No_Cell_Found
to
Radio_Resource
PowerDown_Req
to
Power_Handler
Start_Camping
to
Main_Module
Cell_Selection:
Wait_for_
Received_Levels
Cell_Selection:
Wait_for_
Stop_Deactivate
Cell_Selection:
Init_State
Cell_Selection:
Wait_for_
Synchronized

Results:

- Automatic translation of SDL into SVEL
- Modelling of priority buffers and OS kernel
- Efficiency improvement by manual optimization
- Properties: e.g. deadlock-freeness

$$\mathbf{A}\,\mathbf{G}^{*}\,\mathbf{E}\,\mathbf{F}^{*}\,\texttt{init}$$

No sequence of user actions can bring the phone into a state from where it cannot be reset

- Message buffering is the most critical part
- Priority scheduling could lead to unexpected results

3. Scheduling verification with Spin

SPIN: AT&T model checker for **LTL**

- Author: G. Holzmann, Bell Labs
- Partial order reductions by D. Peled
- On-the-fly state enumeration
- Synchronous communication
- Built-in buffering mechanism
- Main area: protocol verification
- Input language for model: PROMELA
  - CSP-like communication
  - shared variables
  - idling guarded commands

Process synchronization problem in the UTS kernel

```
bit lk, avail, wanted;
mtype = (running, sleeping);
mtype S = running;
active proctype client (){
sleep:
  atomic {(lk == 1) -> lk = 0};
  do
  :: (avail == 0) -> wanted = 1;
     S = sleeping; lk = 1;
     (S == running);
  :: else -> break
  od;
progress:
  assert(avail == 1);
  avail = 0; lk = 1;
  goto sleep }
```

```
active proctype server(){
wakeup:
  avail = 1;
  (lk == 1);
  if
  :: wanted -> wanted = 0;
     if
     :: (S==sleeping) -> S=running;
     :: else -> skip
     fi
  :: else -> skip
  fi;
  goto wakeup }
```

SPIN produces the following counterexample:

Client	Server	Comment
`avail=0`		consume resource
`lk=1`		release lock
`lk==1`		pass lock
`lk=0`		set lock
`avail==0`		resource busy
`wanted=1`		apply for resource
	`wanted==1`	check waiting queue
	`wanted=0`	reset waiting queue
	`else`	no process sleeping!
	`avail=1`	make resource available
`S=sleeping`		go to sleep
`lk=1`		client releases lock
	`lk==1`	server checks lock
	`else`	resource not wanted
	`avail=1`	make resource available
	`lk==1`	server checks lock
	...	...

```
* linux/ipc/sem.c
* Copyright (C) 1995 Eric Schenk, Bruno Haible
*
* IMPLEMENTATION NOTES ON CODE REWRITE (Eric Schenk, January 1995):
* This code underwent a massive rewrite in order to solve some problems
* with the original code. In particular the original code failed to
* wake up processes that were waiting for semval to go to 0 if the
* value went to 0 and was then incremented rapidly enough. In solving
* this problem I have also modified the implementation so that it
* processes pending operations in a FIFO manner, thus give a guarantee
* that processes waiting for a lock on the semaphore won't starve
* unless another locking process fails to unlock.
```

Linux Verification

From the Linux main scheduler function:

```
asmlinkage void schedule(void){
        int c;
        struct task_struct * p, * prev, * next;
        int this_cpu = smp_processor_id();

        c = -1000;
        next = idle_task;
        while (p != &init_task) {
                int weight = goodness(p, prev, this_cpu);
                if (weight > c) c = weight, next = p;
                p = p->next_run; }
        /* if all runnable processes have "counter == 0", re-calculate counters */
        if (!c) { for_each_task(p) p->counter = (p->counter >> 1) + p->priority; }
        next->processor = this_cpu;
        next->last_processor = this_cpu;
        if (prev != next) { kstat.context_swtch++;
                            get_mmu_context(next);
                            switch_to(prev,next)}
        return;
}
```

Abstraction:

- Reduce to finite (fixed) number of processes
- Use finite number of goodness values for process priorities
- Model linked list of processes by SPIN's buffering mechanism

Compositionality:

- Find appropriate specifications for subroutines and functions
- Constrain possible runs by assumptions about the environment

Correctness property: No user process is indefinitely delayed

Ongoing project "LiVE" within the BISS to verify the kernel this way.

4. Deadlock analysis with FDR

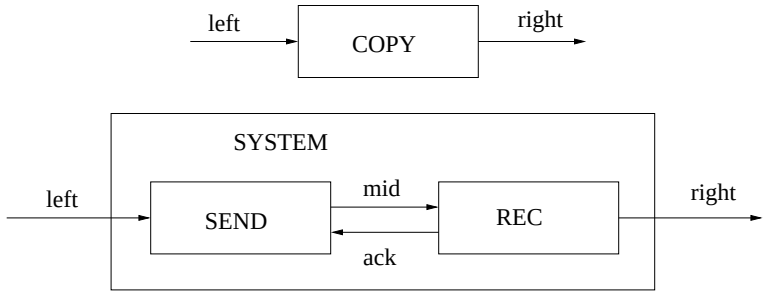
FDR: **F**ailure-**D**ivergence-**R**efinement checker

- Commercial system (Formal Systems Europe Ltd.)
- CSP input language (deterministic transition systems)
- trace inclusion and refinements between processes
- implemented in SML
- efficient transition graph compression

One-directional transmission

$$COPY \triangleq left?x \longrightarrow right!x \longrightarrow COPY$$

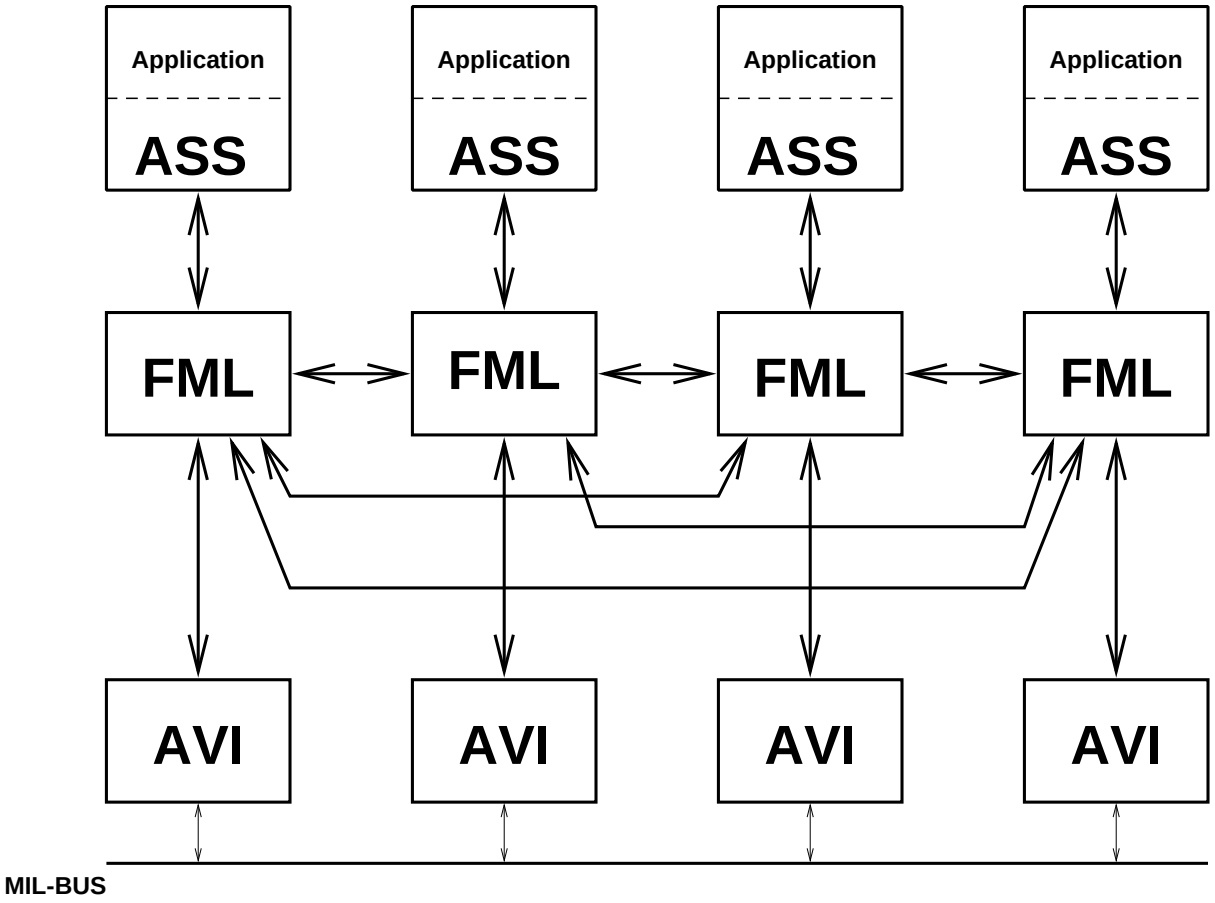
$$SEND \triangleq left?x \longrightarrow mid!x \longrightarrow ack?y \longrightarrow SEND$$
$$REC \triangleq mid?x \longrightarrow right!x \longrightarrow ack!y \longrightarrow REC$$
$$SYSTEM \triangleq (SEND \underset{\{mid,ack\}}{||} REC) \setminus \{mid, ack\}$$



FDR can prove $COPY \sqsubseteq SYSTEM$ and $SYSTEM \sqsubseteq COPY$.

```
-- FDR input for transmission example

-- values to be transmitted
DATATYPE = {apples, oranges, pears}

-- Channel declarations
pragma channel left, right, mid : DATATYPE
pragma channel ack :

-- specification of a single-place buffer
COPY = left ? x -> right ! x -> COPY

-- implementation with two processes
SEND = left ? x -> mid ! x -> ack -> SEND
REC = mid ? x -> right ! x -> ack -> REC
SYSTEM = (SEND [| {| mid, ack |} |] REC ) \ {| mid, ack |}

-- fdr> ParseFile "Filename";
-- fdr> CheckTrace "COPY" "SYSTEM";
```

A fault tolerant computer (DASA DMS-R FTC)

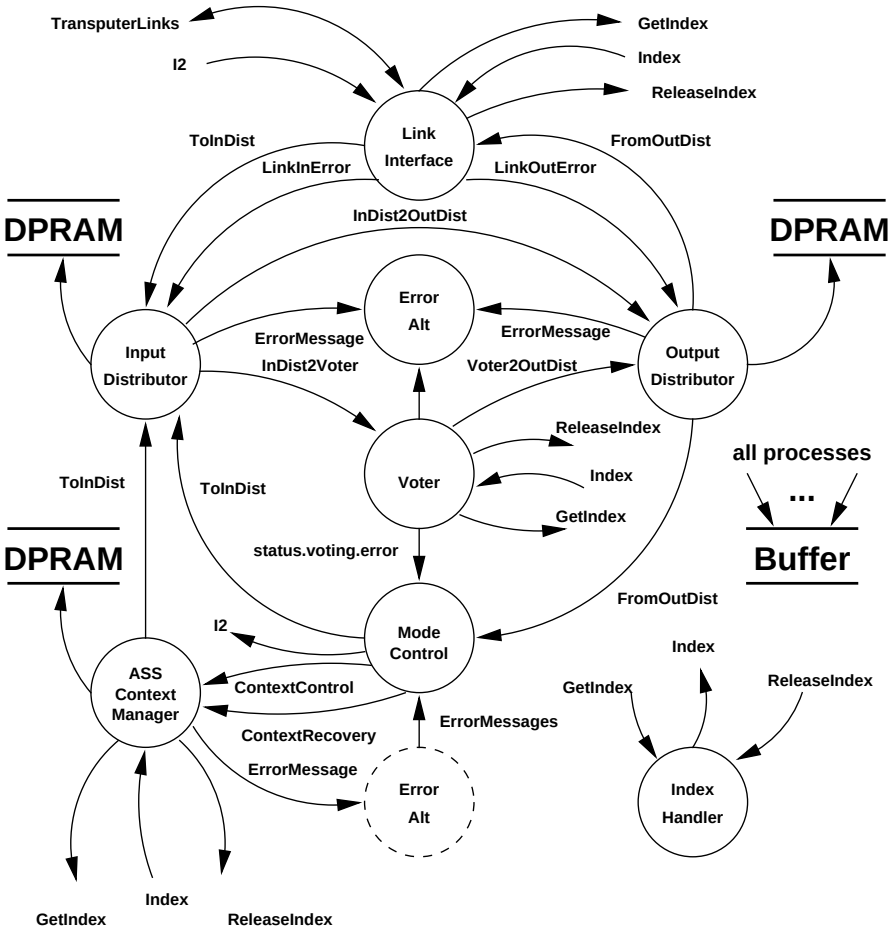


Fault management layer

Available documents:

- Informal description
- Data flow diagrams
- Pseudocode and Occam code

```
IF (asy = OFF) THEN
    store subframe to data pool
    IF (bus.id > 0) THEN
        bus.id = bus.id -1
        output.index = 0
        1. avi enable(standby)
        2. fdir.alloc(fdir.idx)
        ...
```



Deadlock Analysis $\triangleq$

In an environment that always accepts outputs from the system but may or may not refuse to provide inputs, the following holds: Whenever the system reaches a stable state where all communications are blocked, all processes reading from vital channels are ready for input on these channels.

Assumption:
Environment always accepts outputs and sometimes yields inputs.

Reengineering problem
A-posteriori verification (on the finished code) is much more expensive than a-priori verification (during the design phase)!

Abstraction of Occam code

```
P : WHILE TRUE
      SEQ
        IF
          mc = 1
            a!TRUE
          TRUE
            b!FALSE
        c?mc
```

```
Q : WHILE TRUE
      SEQ
        IF
          mc = 1
            a?x
            mc := 0
          TRUE
            b?x
            mc := 1
        c!mc
```

$$P \triangleq (a \longrightarrow c \longrightarrow P) \sqcap (b \longrightarrow c \longrightarrow P)$$
$$Q \triangleq (a \longrightarrow c \longrightarrow Q) \sqcap (b \longrightarrow c \longrightarrow Q)$$
$$S \triangleq P \underset{\{a,b,c\}}{||} Q$$

Abstraction too coarse! (Deadlock in abstraction not in code)

$$
\begin{array}{lcl}
MC & \triangleq & MC1(0) \\
MC1(x) & \triangleq & rp!\,x \longrightarrow MC1(x) \\
 & [\,] & rq!\,x \longrightarrow MC1(x) \\
 & [\,] & wq?\,y \longrightarrow MC1(y)
\end{array}
$$

$$
\begin{array}{lcl}
P & \triangleq & rp?\,mc \longrightarrow \\
 & & \text{if } (mc == 1) \text{ then } a \longrightarrow c \longrightarrow P \\
 & & \text{else } b \longrightarrow c \longrightarrow P \\
Q & \triangleq & rq?\,mc \longrightarrow \\
 & & \text{if } (mc == 1) \text{ then } a \longrightarrow wq!\,0 \longrightarrow c \longrightarrow Q \\
 & & \text{else } b \longrightarrow wq!\,1 \longrightarrow c \longrightarrow Q \\
S & \triangleq & (P \underset{\{c,a,b\}}{||} Q) \underset{\{rp,rq,wq\}}{||} MC
\end{array}
$$

Compositional reasoning

$(P_1 \| \cdots \| P_n)$ is free of deadlock iff
there exist abstract processes $Q_1, \ldots, Q_n$ such that

- $Q_i \sqsubseteq P_i$ for all i, and
- $(Q_1 \| \cdots \| Q_n)$ deadlock free

Heuristics:

- Find "generic theories", i.e., templates for processes that occur frequently in the code and can be shown to be deadlock free.
- Use cycles in data flow to determine possible deadlocks

5. Specifying a controller in STeP

STeP: **S**tanford **Te**mporal **P**rover

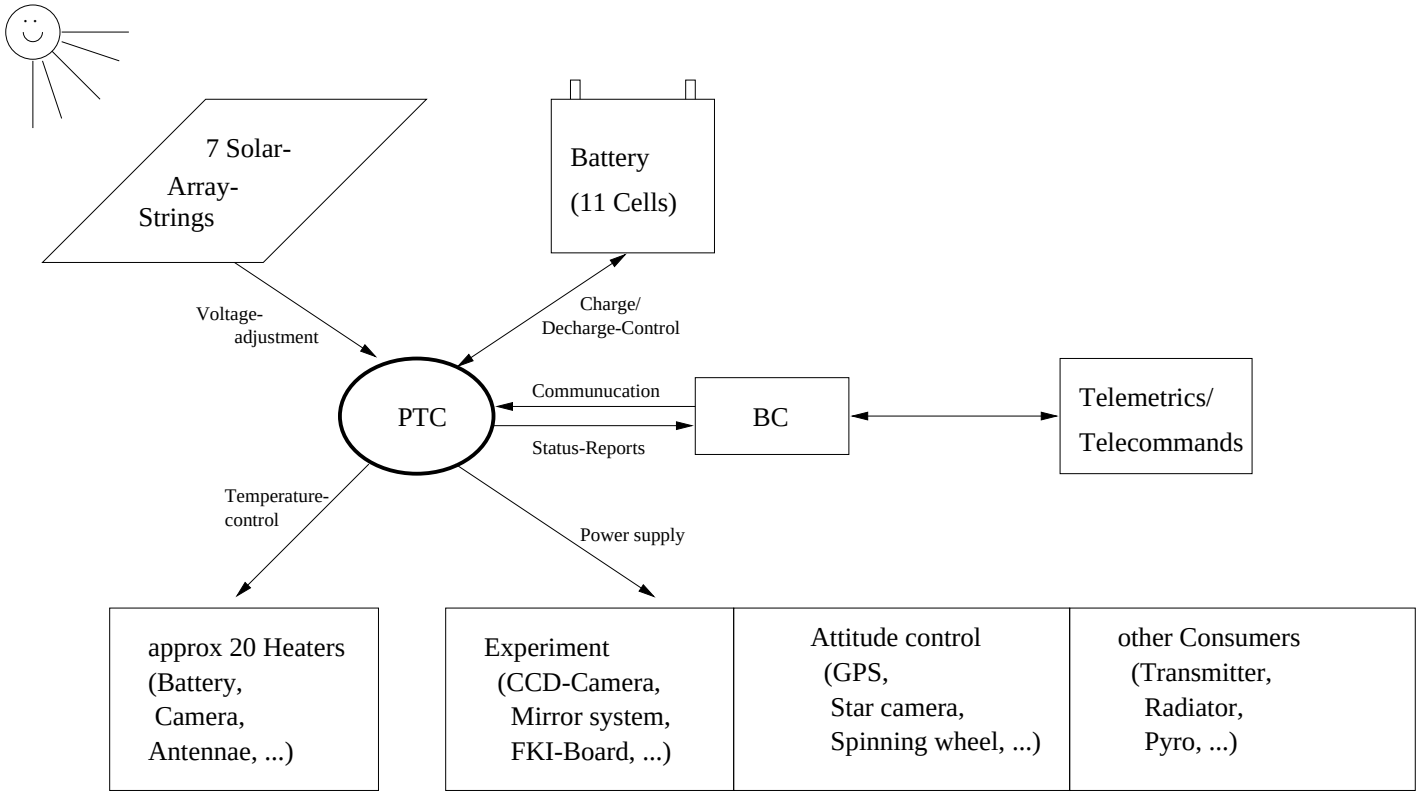
- Manna/Pnueli textbook add-on
- interactive proof system
- BDD model checking component
- efficient depth-first-search
- Simple Programming Language
- **LTL** + first order logic + arithmetic

Example: Distributed binomials

```
in    k, n       : int where ((0 <= k) /\ (k <= n))
local y1, y2, r  : int where y1 = n, y2 = 1
out   b          : int where b = 1

P1:: [ l0: while (y1 > n - k) do (y1,b) := (y1-1, b*y1);
       l6: ]
||
P2:: [ m0: while (y2 <= k) do
         m1: << when (y1+y2) <= n do (y2,b) := (y2+1, b div y2)>>;
       m7: ]

SPEC PROPERTY correctness:
  [](l6 /\ m7 --> ((b * Prod i:[1..k] . i) = (Prod i:[n-k+1..n] . i)))
```

A satellite controller OHB ABRIXAS PTC

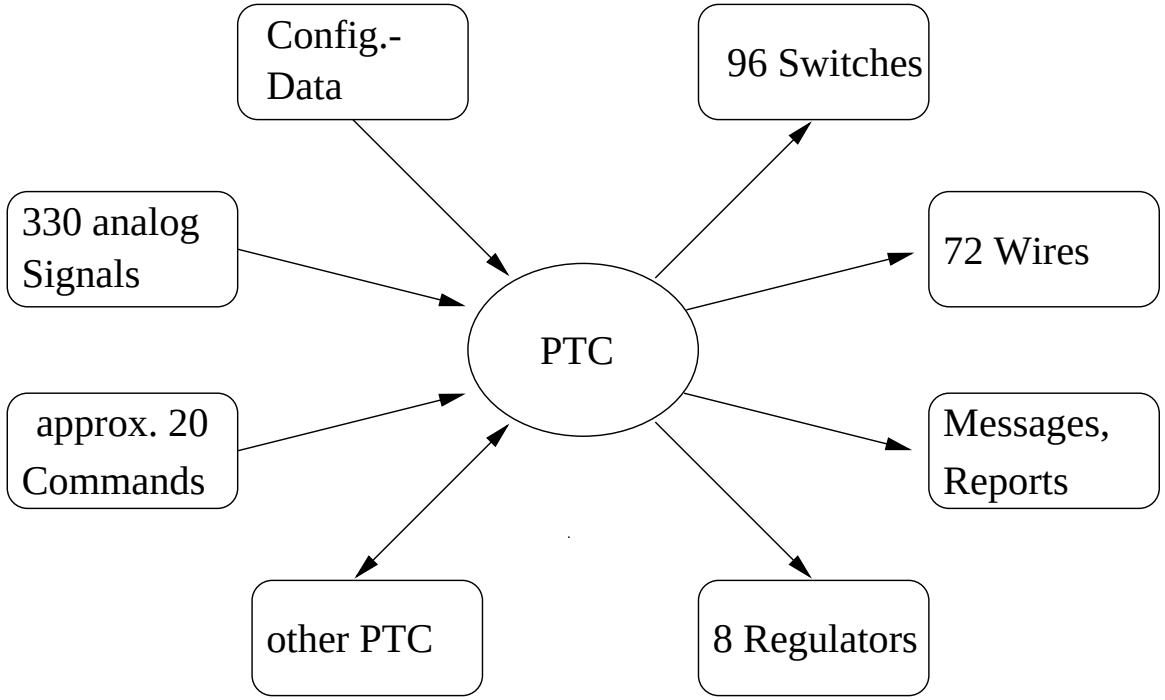
Real-time system for power and thermal control of research satellite



Informal requirements

- When switching any switch, a certain order has to be respected which guarantees that switches are not switched under load
- There must be at least one receiver switched on; switching off both receivers shall be blocked. If a receiver should be on, it must be cyclically reswitched to avoid HW faults. If both receivers are off, a message is generated and both receivers are switched on.
- The battery is charged in two phases: main and trickle charge. Main charge starts at the beginning of the sun phase; a charge current of 6A is to be supplied which shall flow until the capacity discharged during the last shade phase is recharged. Then it must be controlled whether a certain minimal battery pressure is reached; the charge current is to be kept constant as long as this pressure is not reached. When the minimal pressure is reached, main charging stops.

Formalization of interfaces



Reformulation of properties

Semiformal requirements discussed with the developers:

- Switch `Swt_EXP_PWR_RED` is never switched while `Swt_EXP_PWR_MAIN` has been switched on.
- If at least one of both receivers is fuctional, at least one is on.
- During the sun phase the charge current must be `I_Charge`, until the discharged amount is recharged and the minimal pressure `P_minabs` is reached.

Temporal logic formulation

- $\mathbf{G}^*\,(\texttt{EXP\_PWR\_MAIN} = \texttt{ON} \rightarrow \neg\texttt{Swt\_EXP\_PWR\_RED})$
- $\mathbf{G}^*\,((\texttt{REC1\_PWR} = \texttt{ON} \vee \texttt{REC2\_PWR} = \texttt{ON})\ \mathbf{W}^+\ (\texttt{REC1\_FAIL} \wedge \texttt{REC2\_FAIL}))$
- $\mathbf{G}^*\,(\texttt{Tau\_SUN\_ON} \rightarrow \mathbf{F}^+\,((\texttt{I\_Charge} - \delta \leq I \leq \texttt{I\_Charge} + \delta)\ \mathbf{U}^+\ (\texttt{recharged} \wedge \texttt{P\_BATT\_PRESS\_OK})))$

A number of incompletenesses and inconsistencies was discovered that way.
For various reasons, a complete verification with STeP seemed not feasible:

- Shooting at a moving target
- High dependency on configuration tables and hardware
- Real-time properties hard to verify

CSP reformulation

```
CHARGE_CONTROL = Tau_SUN_ON -> setTimChargeControl ->
                 MAIN_CHARGE(false, false)

MAIN_CHARGE(P_BATT_PRESS_1_OK, recharged) =
    (Evt_I_BATT_MAIN_IS_I_Charge -> resTimChargeControl ->
                 MAIN_CHARGE(P_BATT_PRESS_1_OK, recharged))
 [] (elaTimChargeControl -> errorChargeControl -> setTimChargeControl ->
                 MAIN_CHARGE(P_BATT_PRESS_1_OK, recharged))
 [] (Evt_I_BATT_MAIN_ISNOT_I_Charge -> setTimChargeControl ->
                 MAIN_CHARGE(P_BATT_PRESS_1_OK, recharged))
 [] (Evt_P_BATT_PRESS_1_OK ->
       if recharged then SUPP_CHARGE
                    else MAIN_CHARGE(true, recharged))
 [] (Evt_recharged ->
       if P_BATT_PRESS_1_OK then SUPP_CHARGE
                            else MAIN_CHARGE(P_BATT_PRESS_1_OK, true))
 [] (Tau_SUN_OFF -> DISCHARGE_CONROL)
```

Formal Testing

- Test sequences automatically generated from specifications
- Automatic hardware-in-the-loop test execution
- Exhaustive testing converges to complete model checking

Results

- Errors mainly under exceptional circumstances
- Formalization lead to design improvement
- Verification and testing complementary methods

Summary

- Model checking for "real systems" feasible
- Cost efficient quality improvement
- Maturing from "black art" to "solid craft"

Have a nice conference!