Branch & Bound with SPIN 4.0

and the Cybernetix Case Study

University of Twente, The Netherlands

AMETIST meeting, Dortmund

Monday 2-Dec-2002

Theo C. Ruys

University of Twente
Formal Methods & Tools group
http://www.cs.utwente.nl/~ruys

Work in Progress

Overview

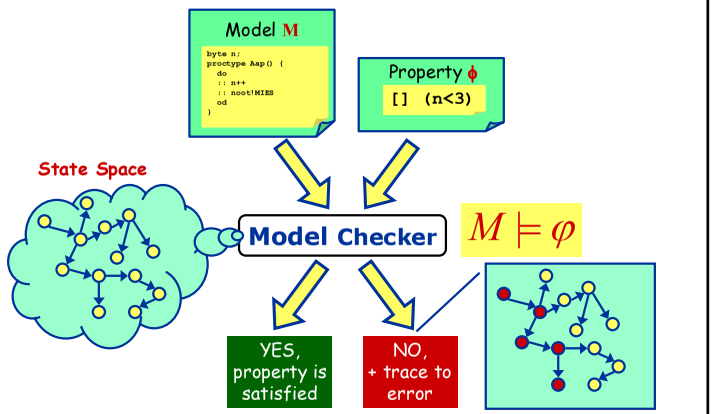
- Introduction
- Model checking and scheduling problems
- Soldiers / Hippies problem
- SPIN 4.0 – new features
- Hippies problem with SPIN 4.0
- Cybernetix case study with SPIN 4.0
- Conclusions

Background

- [Ruys & Brinksma – TACAS 1998]
  - introduces the soldiers problem
  - SPIN is used to solve this simple scheduling problem
- SPIN Cookbook
  - recipes for using SPIN in an effective way
  - "dessert" – Travelling Salesman Problem (Oct. 1999)
    - used the same approach as [Ruys & Brinksma 1998]
    - exploited bitstate hashing to tackle "large" TSPs
- AMETIST Meeting (Sep. 2002)
  - Attempts to attack the "Cybernetix Case", a typical job shop scheduling problem.

We had the feeling that we could do better with SPIN.

What is Model Checking?



SPIN - Introduction (1)

SPIN (= Simple Promela Interpreter)

= is a tool for analysing the logical conisistency of concurrent systems, specifically of data communication protocols.

= state-of-the-art model checker, used by >2000 users

– concurrent systems are described in the modelling language called Promela.

SPIN - Introduction (2)

Promela (= Protocol/Process Meta Language)
– specification language to model finite-state systems
– loosely based on CSP
  • dynamic creation of concurrent processes
  • communication via message channels can be
    – synchronous (i.e. rendezvous), or
    – asynchronous (i.e. buffered)
– features from Dijkstra's guarded command language
– features from the programming language C

Promela is a modelling language, not a programming language.

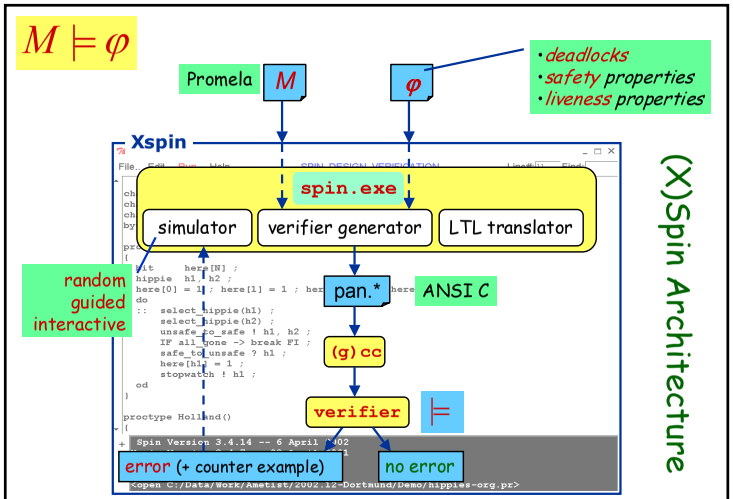
SPIN - Introduction (3)

- Major versions:

1.0	Jan 1991	initial version [Holzmann 1991]
2.0	Jan 1995	partial order reduction
3.0	Apr 1997	minimised automaton representation
4.0	late 2002	embedding of C code

- Some success factors of SPIN
  – "press on the button" verification (model checker)
  – very efficient implementation (using C)
  – nice graphical user interface (Xspin)
  – not just a research tool, but well supported
  – contains more than two decades research on advanced computer aided verification (many optimization algorithms)

1983 Unix
1986 TeX
1991 TCP/IP
1997 Tcl/Tk



Model checking "scheduling problems" (1)

- M = model of the problem in Promela
  - with costs (or time) added to states/transitions
  - a variable `cost` is updated when a transition is taken or a state is reached.
- Goal: find schedule to an end-state with minimum cost.
  1. Verify that M is error-free.
  2. Find optimal schedule:

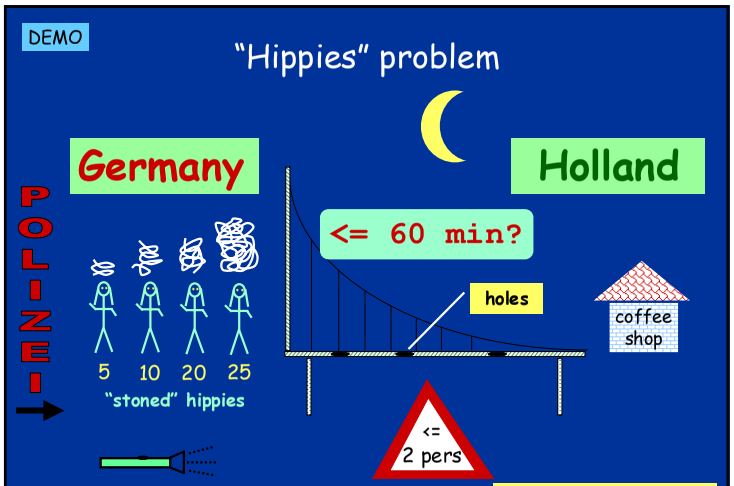
"eventually cost will be larger than max"

```
max := guess of (worst case) maximum cost
do
   verify M |= (<> cost > max)
   if (error)
   then max := cost
while (error)
```

If there is a path to a final state for which the cost is less than max, SPIN will generate an error trail leading to this state.

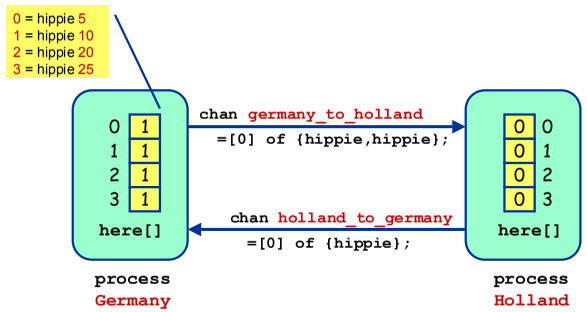
DEMO

"Hippies" problem



Original "soldiers" problem, see: [Ruys & Brinksma 1998]

Modelling the "hippies" problem



It is clear that you want to send two hippies over to Holland but only one hippie back with the flashlight.

DEMO

Solution to the Hippies problem (1)

```
chan germany_to_holland = [0] of {hippie, hippie} ;
chan holland_to_germany = [0] of {hippie} ;
chan stopwatch          = [0] of {hippie} ;
byte time ;
...
proctype Germany()
{
  bit    here[N] ;
  hippie h1, h2 ;
  here[0]=1; here[1]=1; here[2]=1; here[3]=1;
  do
  :: select_hippie(h1) ;
     select_hippie(h2) ;
     germany_to_holland ! h1, h2 ;
     IF all_gone -> break FI ;
     holland_to_germany ? h1 ;
     here[h1] = 1 ;
     stopwatch ! h1 ;
  od
}
```

the "cost"

A hippie is a byte.

Process "Holland" is the dual of "Germany".

A hippie is randomly chosen from the hippies that are still "here".

It can be modelled more effectively See [Ruys 2001] for directions.

DEMO

Solution to the Hippies problem (2)

```
proctype Timer()
{
end:
  do
  ::  atomic { stopwatch ? 0 -> time=time+5 ; MSCTIME }
  ::  atomic { stopwatch ? 1 -> time=time+10; MSCTIME }
  ::  atomic { stopwatch ? 2 -> time=time+20; MSCTIME }
  ::  atomic { stopwatch ? 3 -> time=time+25; MSCTIME }
  od
}

init {
  atomic { run Germany(); run Holland(); run Timer(); }
}
```

Now we should check:
◊(time>60)

Model checking "scheduling problems" (2)

- Idea of using model checkers for solving scheduling problems has been taken up. See for instance:
  - [Brinksma & Mader - SPIN 2000]
  - [Larsen et. al. – CAV 2001]
  - [Ansgar Fehnker - PhD Thesis 2002]

Model Checkers are being used for serious scheduling problems!

- Original idea works, but is inefficient:
  - the (initial) complete state space already contains the most optimal solution;
  - iteratively checking ◊ (cost > max) to obtain this solution is not needed, of course.

However, due to SPIN's on-the-fly model checking algorithm, for each subsequent iteration, less of the state space has to be checked: SPIN stops when it finds a state for which `cost>max` holds.

Branch & Bound

- Branch & Bound (from "operations research", early sixties)
  - is an approach developed for solving discrete and combinatorial optimisation problems, especially integer programming problems
- Idea:
  - enumerate all possible solutions
  - represent these solutions by a directed graph/tree
    - leaves are end-points of possible solutions
    - a path from the starting node to a leaf represents a solution
  - while building the graph (= state space), we can stop considering the descendents of an interior node, if
    - it is certain that all paths via this node will lead to an invalid path, or
    - if is certain that all paths via this node will have higher cost than the best path found so far.

The branch-and-bound approach is not a heuristic, or approximating procedure, but it is an exact, optimising procedure that finds an optimal solution.

SPIN 3.x

- If we could make the best cost "global" to all execution paths, we could update this best cost each time we find a better one.
- Sketch to find an optimal schedule in SPIN version 3.x:
  - Add a global variable `best_sofar` to `pan.c` that is global for all verification runs.
  - Everytime a correct schedule is found of which the cost is lower, the variable `best_sofar` is updated and the trail leading to this schedule is saved.
  - The variable `best_sofar` will not be part of the state vector, but will be global to the verification.
  - The variable `best_sofar` is initialised in a special section before the verification is started.

Drawback: we have to change the source code of SPIN.

SPIN 4.0 (1)

Alpha version of SPIN 4: http://spinroot.com

- SPIN 4.0 supports the inclusion of embedded C code into Promela models. Five new primitives:

`c_decl { … }`	to introduce C types that can be used in the Promela model
`c_state …`	to add new C variables: Global, Local or Hidden
`c_expr { … }`	to execute a C expression and use the return value in the model
`c_code { … }`	to add atomic C statements to the model (e.g. printing)
`c_track { … }`	can be used to track memory, holding state information

SPIN 4.0 (2)

- The purpose of the new primitives is to provide support for automatic model extraction from C code.
  - to build your "own" FeaVer [Holzmann 2000] .
- … "The capability to embed arbitrary fragments of C code into a Promela model is very powerful and therefore easily misused" …

But we can safely use it to find the optimal solution for a scheduling problem, like the "hippies" problem.

Another feature of SPIN 4.0: `pan` now has a "guided simulation" mode. It is not longer needed to replay the simulation with `spin`.

DEMO

hippies-o1.pr

Just printing the time of the schedules found.

```
proctype Holland()
{
  bit     here[N] ;
  hippie  h1, h2 ;
  do
  ::  unsafe_to_safe ? h1, h2 ;
      here[h1] = 1 ;
      here[h2] = 1 ;
      stopwatch ! max(h1, h2) ;
      IF all_here -> break FI ;
      select_hippie(h1) ;
      safe_to_unsafe ! h1
  od ;
  c_code {
      printf("found another schedule: %d\n", now.time);
  }
}
```

"now" is the current state; we can access the global and local variables in this state.

we can even save the schedules by calling pan's `putrail()`

DEMO

hippies-o2.pr

Only saving the best schedule.

- In the declaration part of the Promela file:

```
#define MAX_TIME    1000
c_state "int  best_time" "Hidden"
```

This declaration is copied verbatim to pan.c.

"Hidden" means that the variable is not stored in the state vector, but is global for the whole verification.

- At the end of the `Holland` proctype (i.e. in the final state of a possible schedule):

```
c_code {
    if (now.time < best_time) {
        best_time = now.time;
        printf("> best time now: %d\n", best_time);
        putrail();
        Nr_Trails--; /* only save the best trail */
    };
}
```

The variable "now" corresponds with the current state.

`putrail` and `Nr_Trails` are globals of `pan.c`.

- Initialise `best_time` in `init`:

```
c_code { best_time = MAX_TIME; };
```

DEMO

hippies-o3.pr (1)

Optimisation: simple (branch &) bound.

- Simple optimisation:
  - If after the "stopwatch has been pressed", the **time** is already greater than **best_time**, we know that this execution trace will not lead to a better trace. So, we stop searching the state space.

Note: this scheme *can* be improved upon!

- In the **Germany** proctype:

```
...
stopwatch ! h1 ;
IF c_expr { (now.time > best_time) } -> break FI;
```

Beware: we are now changing the model; invalid end-states will get introduced!

- In the **Holland** proctype:

```
...
stopwatch ! max(h1, h2) ;
IF c_expr { (now.time > best_time) } -> break FI;
```


DEMO

hippies-o3.pr (2)

Optimisation: simple (branch &) bound.

The complete **Holland** proctype would then become:

```
proctype Holland()
{
  bit     here[N] ;
  hippie  h1, h2 ;
  do
  ::  unsafe_to_safe ? h1, h2 ;
      here[h1] = 1 ;
      here[h2] = 1 ;
      stopwatch ! max(h1, h2) ;
      IF c_expr { (now.time > best_time) } -> break FI;
      IF all_here -> break FI ;
      select_hippie(h1) ;
      safe_to_unsafe ! h1
  od ;
}
```


DEMO

hippies-o4.pr

Optimisation: (branch &) bound in never claim.

- Branch & bound in never claim:
  - Instead of pruning the search tree in the Promela model, we can also limit the search of the state space via a **never** claim (i.e. combination with original idea).

- We *on-the-fly* check <> (now.time >= best_time):

```
#define higher_cost (c_expr { now.time >= best_time })

never {     /* !<> higher_cost */
    skip;   /* skip over c_code { best_time ...; } of init */
accept_init:
T0_init:
    if
    :: (! (higher_cost)) -> goto T0_init
    fi;
}
```

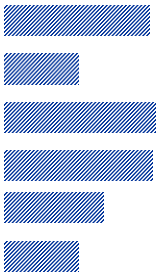
Note that the property we are checking is altered during the verification.

SPIN verifies a LTL formula using a never claim, that is automatically generated from the formula.

Hippies - results

Note: the max value is not known, typically.

filename	*description*	*# states*
original	deadlock checking	**2541**
original	<> (time >= 60)	**1325**
hippies-o1	just print all schedules found	**2659**
hippies-o2	saving the best schedule	**2630**
hippies-o3	branch & bound in processes	**1766**
hippies-o4	branch & bound in never claim	**1330**



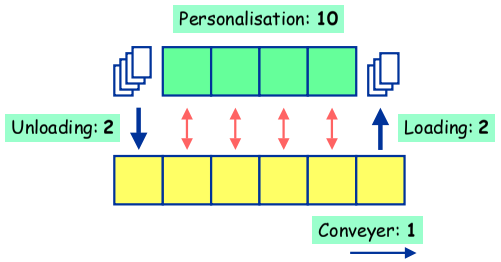
Summary (so far)

- With SPIN 4.0, we can now influence the traversal of the state space (i.e. the verification):
  - in the Promela model
  - but (even more conveniently) in the property
- ... and for debugging purposes we can now also add `printf`-statements to the verification process.

Cybernetix Case Study

- Smart Card Personalisation Machine:
  - takes a pile of the blank smart cards
  - programs them with the personalised data
  - prints them and
  - tests them
- Order of the original cards should be preserved.
- Objectives of the case study
  - synthesis of correct and optimal schedules.
  - come up with a general model and tool for Cybernetix.

Model of the Personalisation Machine



- Constraints / Assumptions:
  - The Conveyer cannot move during Unloading and Loading.
  - Starting the Personalisation is instant.
- Parameters:
  NPERS: number of stations
  NCARDS: number of cards

Cybernetix Model in SPIN (1)

initial model based on the Uppaal model of [Krilavičius & Usenko]

Global variables: NCELLS == NPERS+2

```
short belt[NCELLS];      /* conveyer belt  */
```

```
belt[i] == 0:   slot on belt is empty
belt[i] >  0:   card is not yet personalised
belt[i] <  0:   card is already personalised
```

```
short time;              /* time elapsed  */
```

The Conveyer, Unloader and Loader processes update the `time`; the Pers(onalization) process waits till the `time` has elapsed.

Cybernetix Model in SPIN (2)

```
proctype Unloader() {
    /* Unloads a card onto the conveyer belt. */

    short nextCard = 1;   /* next card to be put onto the belt */

    do
    ::  d_step { (nextCard <= NCARDS) && (belt[0] == EMPTY) ->

          belt[0]  = nextCard;
          nextCard = nextCard+1;
          time     = time + tUNLOAD;

          printf("> Unloader (time + %d => %d)\n", tUNLOAD, time);
          PRINT_BELT;
        }
    od
}
```

Only present for visualizing a schedule found; these print-statements will not be executed during the verification.

Cybernetix Model in SPIN (3)

```
proctype Pers(int i) {
    short card, finish_time;

    do
    ::  d_step { (belt[i] > 0) ->
          card        = belt[i];
          belt[i]     = EMPTY;
          finish_time = time + tPERSONALISE;

          printf("> Pers. station %d STARTED: card %d (time now: %d) \n",
                 i, card, time);
          PRINT_BELT;
        }

        d_step { (finish_time <= time) && (belt[i] == EMPTY) ->
          belt[i]     = -card;
          finish_time = 0;

          printf("> Pers. station %d ENDED:   card %d (time now: %d) \n",
                 i, card, time);
          PRINT_BELT;
        }
    od
}
```

To allow more optimisations, in the current model, these variables are now global.

`Pers(i)` waits till the time has elapsed and it can place the personalised card back onto the belt.

Cybernetix Model in SPIN (4)

We also need a process to "tick" away the time for the Personalisation stations. In other words: the conveyer and unloader/loader sometimes have to wait for a personalisation station to be finished.

```
#define MAX_TICKS 30

proctype Tick() {
    byte n;
    do
    ::  d_step { (n < MAX_TICKS) ->
          time = time + 1;
          n    = n + 1;
          printf("> Tick    1 (%d)\n", time);
        }
    ::  else -> break
    od
}
```


Cybernetix Model in SPIN (5)

- This first, naïve model did work, but was not efficient enough.
  - With SPIN we (roughly) got the same results as [Krilavičius & Usenko].
- First (obvious) improvements:
  - Unloading & loading can happen at the same time, so both processes should be merged.
  - Using `d_step`'s instead of `atomic`'s.
  - Resetting all local variables to zero after using them in a computation.

Optimisations

- Optimisations in the model:
  - (++) `TimeAdvance`() instead of `Ticks`()
    - Following [Brinksma & Mader 2000], we let the time jump to the next moment in time when a new personalised card is ready.
  - (+) Changing the scheduling of the processes such that a (semi) optimal schedule is found fast.
  - (+) Only allowing Personalisation to start directly after a Conveyer move.
- Bounding the search:
  - (+) Bounding the search on incorrect schedules.
  - (++) Check if the minimum finish time for the last card (i) in the unloader, (ii) on the belt or (iii) in a personalisation station is already greater than the `best_time` so far.

Bounding the search

- All bound optimisations are taken care of in the never claim:

```
#define higher_cost (c_expr { \
    (now.time >= best_time) || \
    (now.time < 0) || \
    (will_not_be_faster()) || \
    (! schedule_ok()) \
})

never {    /* !<> higher_cost */
    skip; /* skip over 1st step of the init-process */
accept_init:
T0_init:
    if
    :: (! (higher_cost)) -> goto T0_init
    fi;
}
```

bound on current time

the model sets time to -1 for invalid schedules

checks the most optimal time for the last card

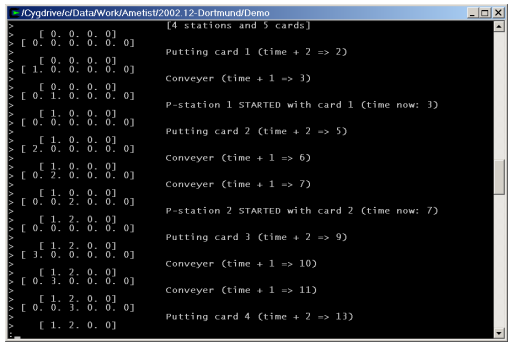
checks if the current path can still lead to a valid schedule

`will_not_be_faster()` and `schedule_ok()` are ordinary C functions that use the current state "`now`" and the `best_time`.

DEMO

Cybernetix - DEMO

`NPERS=4 NCARDS=5`



Some observations

- The state space is more breadth than deep!
- Due to SPIN's state matching, SPIN will always find only one optimal schedule!
  - An extra variable in each state is needed to make it unique.
- If the personalisation time is low w.r.t. the number of stations, the optimal schedule will not use all personalisation stations.
  - Example: **`NPERS=4 NCARDS=4 tPERSONALISE=5`**
    - 108 different schedules that need 27 time units.
    - Only a single one for the optimal schedule in 26 time units.
    - 55x10^5 states to find all possible schedules (only 2x10^5 to find the optimal one)
    - The optimal schedule only uses two of the personalisation stations.

Effectiveness of optimisations

`NPERS=4 NCARDS=4`
`SPIN: -DBITSTATE`

		# states	# secs
v1	initial model using `Tick()`	46325800	1080.8
v2	= v1 + `TimeAdvance()`	2364370	49.5
v3	= v2 + better scheduling of procs	1739010	34.9
v4	= v3 + `<> .. || !schedule_ok()`	1345000	26.2
v5	= v3 + `<> .. || will_be_too_late()`	93199	2.2
v6	= v4 + v5	65999	1.6
v7	= v6 + get all schedules (+ magic number)	953866	15.7

Comparison with [Krilavičius & Usenko]

"standard" Uppaal (Spring 2002) – no cost-reward
[Krilavičius & Usenko] Uppaal 2k
Sun/Solaris: 1Gb RAM

vs.

`-DBITSTATE`
SPIN 4.0α, P3 Mobile/1Ghz
Windows 2000, 256Mb RAM

Cards / Stations	2	3	4	5	6	7	8	9
2 (4 cells)	20	28	32	39	43	50	54	61 251245 4.1 sec
3 (5 cells)	21	25	30	34	38	42 579097 8.8 sec	47 1.9 x 10^6 30 sec	51 3.7 x 10^6 59 sec
4 (6 cells)	22	27	30	35	39 3.6 x 10^6 64 sec	43 13 x 10^6 231 sec	47 34 x 10^6 635 sec	50 56 x 10^7 1057 s.

SPIN 4.0 computed optimal schedules for all entries in the 2-4 x 2-9 table within 36 minutes.

Comparison with Uppaal

- Compared to the timed-approach of [Krilavičius & Usenko] with standard Uppaal, the tuned approach with SPIN
  - performs (much) better
  - could even be better if we would use the time of the optimal schedule (super single mode) to find the schedule
- Disadvanteges/advantages of using SPIN:
  - you have to add time yourself
  + more information on and during the verification process
  + more ways to "guide" the search
  + possible to get all possible schedules (as "error" trails)
  + SPIN's optimisation options, e.g. bitstate hashing
- What about a "special purpose cost/reward SPIN"?
  - This could be as expensive as the "magic number" trick to get all possible schedules.

Discussion

- Job-shop scheduling (and TSP) is NP-complete.
  - The MC-approach (even with branch-and-bound) seems counter intuitive.
    - the state space is enormous, but still we set out to generate (part of) this state space
  - Does the MC-approach scale up?
    - TSP problems fail around N==20
    - Cybernetix case study: only toy-parameters
  - Approximation algorithms for NP-hard problems
    - try to find a "good" solution as soon as possible
    - improve the solution locally
    - does not guarantee to find the "optimal" solution
  - The MC-approach might not always give an answer…

Conclusions

- However, the MC-approach is still appealing:
  - Use the model checker to verify that the formalisation of the problem is correct.
  - Use the model checker to generate an (optimal) solution to the problem.
- SPIN 4.0 offers nice features to implement the Branch & Bound approach on the Promela level.
  - Branch & Bound functionality can nicely be isolated in the ◊-property (i.e. the never claim).
    - which is also less costly than implementing it in the model
- SPIN 4.0 seems to outperform "standard" UPPAAL for discrete-time scheduling problems.

Future work

- Improvements of the Cybernetix model:
  - trying to decide earlier on non-valid schedules;
  - even better "bound" functions (e.g. using the known optimal times for sub-problems);
  - garbage collection of the state space after finding new optimal solutions;
- Future directions:
  - add defective cards
  - add flip-over stations
  - checking the "optimal cycle" in the schedule
  - alternative architecture of machine [Angelika Mader]