Monadic Program-based Tests

An Exercise in Test and Proof

Burkhart Wolff
Univ - Paris-Saclay

Philosophical Statement: Formal Testing

- I know, Testing has for quite a few people a bad name
- Dijkstra ` s Verdict, although misleading and deceitful, did a lot of damage to discredit testing as a verification technique (albeit standards on SE look this oppositely)
- The Science of Testing is as important to The Science of Computing as

  Experiments are to Physics.

Philosophical Statement: Formal Testing

- Formal Testing is defined by A Test Generation Procedure with the following properties:
  - Input: a formal, semantic Model M, a program P, and a coverage criterion CC
    (a test generation procedure does not necessarily take all three into account)
  - Output: Generate test-data (and entire test environments taking an environment into account)
  - Ideally: Generation approximates Exhaustive Verification

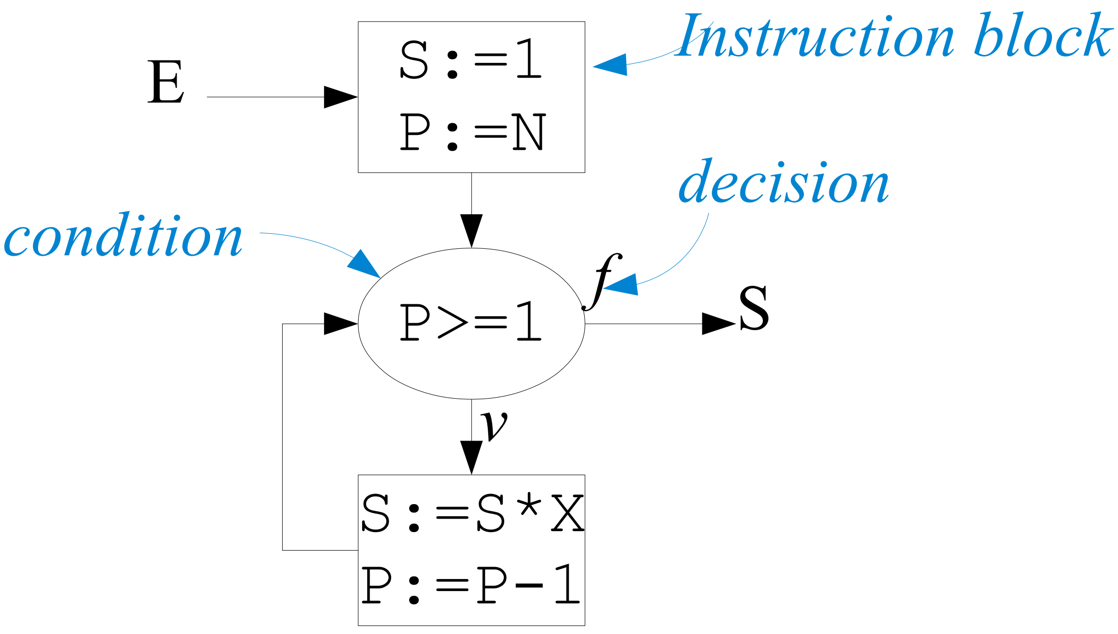
Program-based Testing

- From the wealth of test-generation procedures and methods, we chose a classic: program-based testing (a la Pex, Path-Crawler, to a certain extent: SAGE).
- Idea:
  - Convert the program into a CFG,
  - draw execution paths according your CC,
  - calculate path-expressions for the chosen parts,
  - use constraint-solvers to construct input
  - Run input on program and check post-cond.

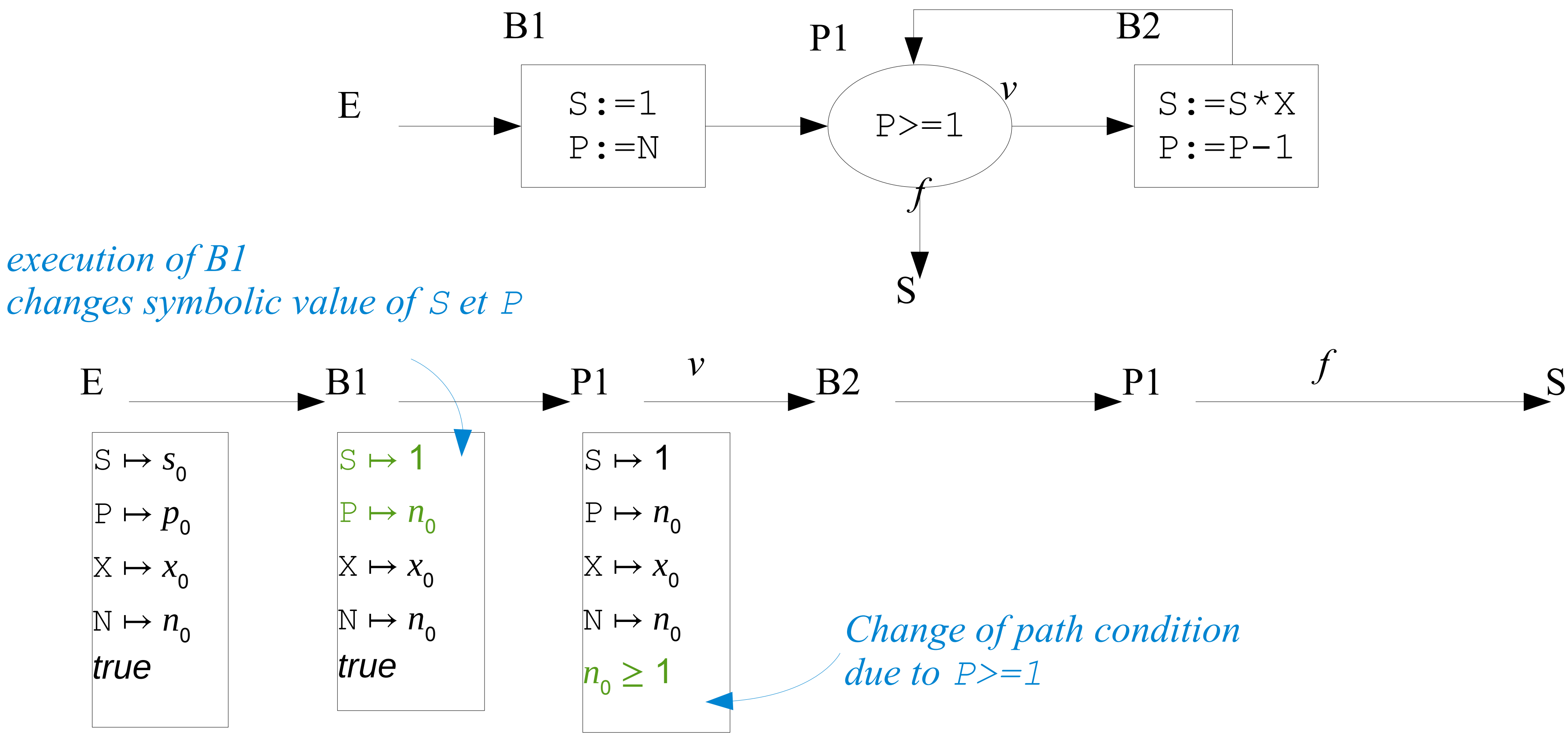
Phase I

Program to CFG

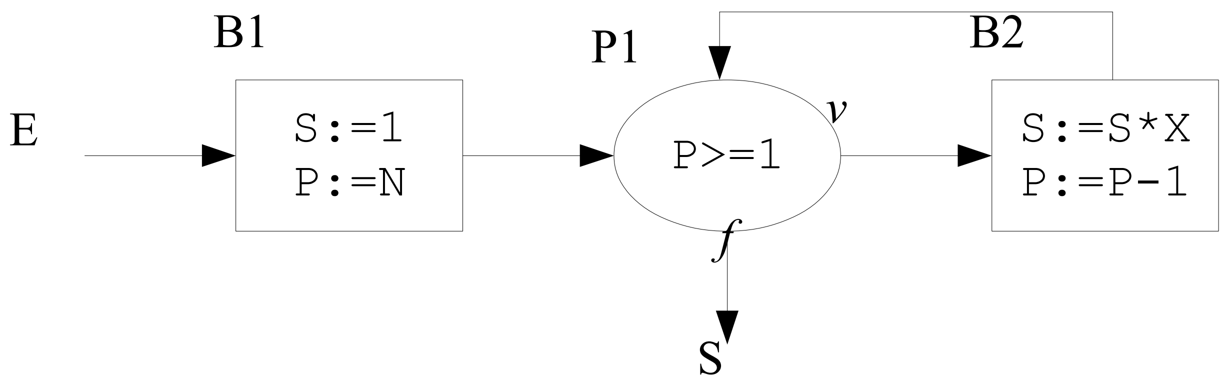
```
S:=1;
P:=N;

while P>=1
do

   S:=S*X;
   P:=P-1;

endwhile;
```

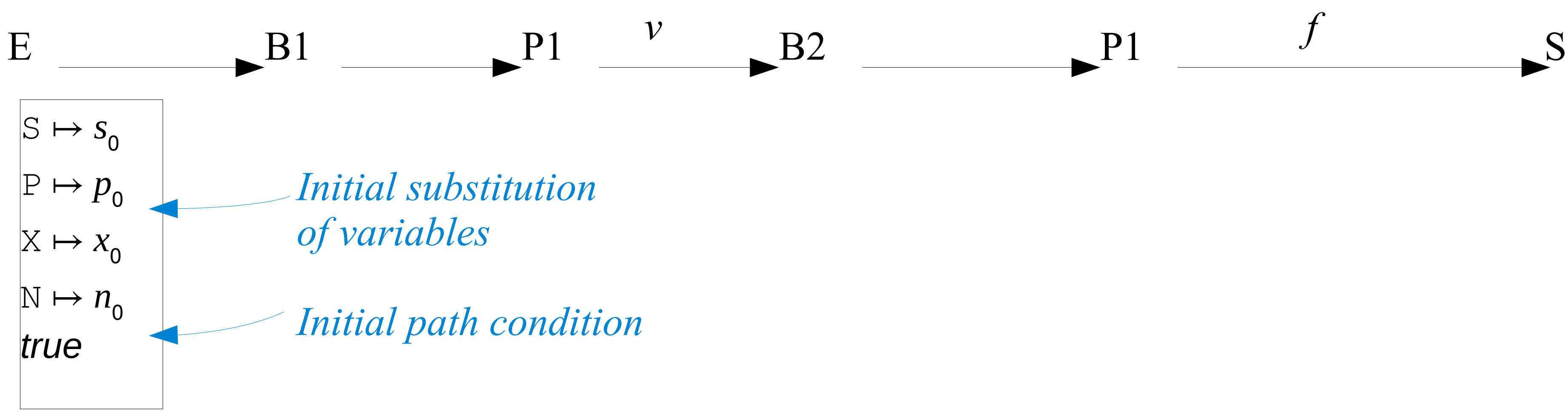


Phase II: Formal symbolic Execution

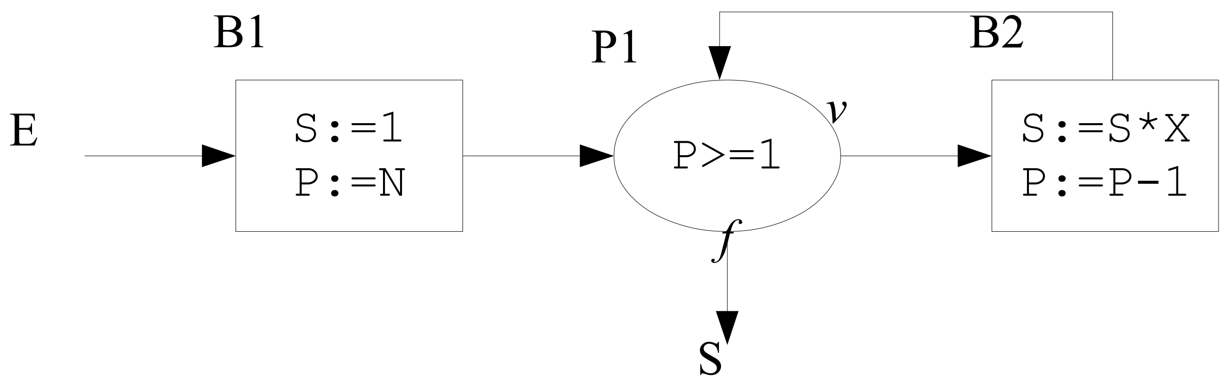


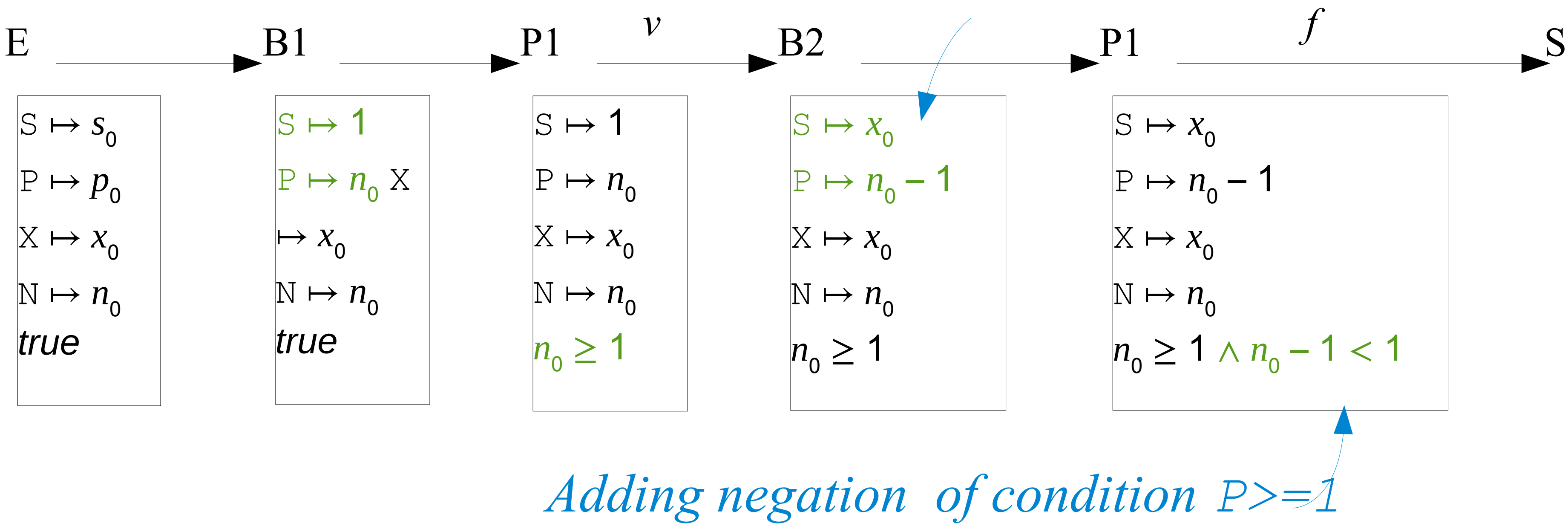
Phase II: Formal symbolic Execution



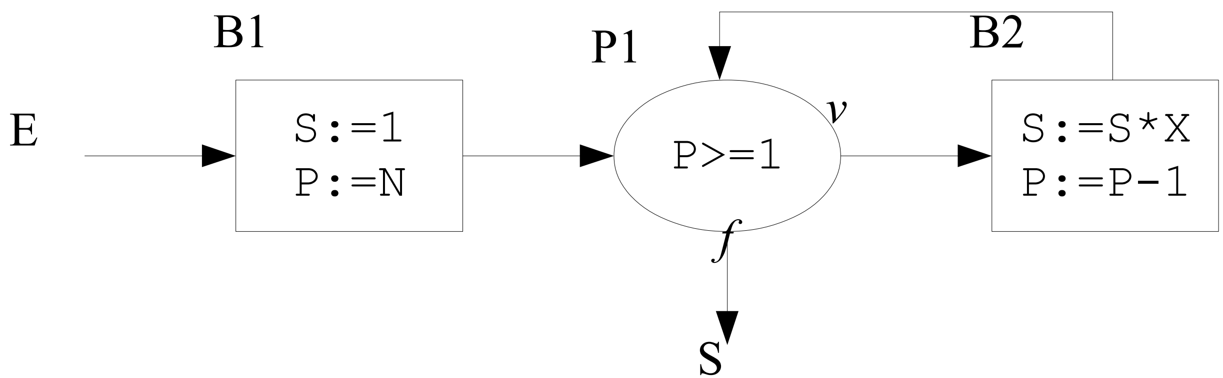


Phase II: Formal symbolic Execution





Phase II: Formal symbolic Execution



E →	B1 →	P1 —v→	B2 →	P1 —f→ S
S ↦ s_0	S ↦ 1	S ↦ 1	S ↦ x_0	S ↦ x_0
P ↦ p_0	P ↦ n_0 X	P ↦ n_0	P ↦ $n_0 - 1$	P ↦ $n_0 - 1$
X ↦ x_0	↦ x_0	X ↦ x_0	X ↦ x_0	X ↦ x_0
N ↦ n_0	N ↦ n_0	N ↦ n_0	N ↦ n_0	N ↦ n_0
true	*true*	$n_0 \geq 1$	$n_0 \geq 1$	$n_0 \geq 1 \wedge n_0 - 1 < 1$

Final Path Condition : $n_0 \geq 1 \wedge n_0 - 1 < 1 \Leftrightarrow n_0 = 1$

Test and Then ?

- Phase III : Constraint solving (trivial here)
- Phase IV : Test Execution (satisfies the result of the program run the post-condition ?)

Is there a more direct, elegant way to represent and Reason over Program-based Tests than this procedure ?

Yes, use Monads …

Introduction to Sequence Testing

- Some notions of traditional sequence testing
  - *Input-output tagged Partial Deterministic Automata (IOPDA),*

    e.g. $A = (\sigma, \tau::(\sigma \times (\iota \times o) \Rightarrow \sigma\ option))$,
    - σ is the type of states
    - ι the type of inputs (input events)
    - o the type of outputs (output events)
    - τ the set of input-output-transitions.

Introduction to Sequence Testing

- Some notions of traditional sequence testing
  - *Input-Output Automata (IOA),*
    e.g. $A = (\sigma, \tau{::}(\sigma \times (\iota + o) \times \sigma)\text{set})$,
    - σ is the type of states
    - ι the type of inputs (input events)
    - o the type of outputs (output events)
    - τ the set of input-output-transitions.

How to model and test stateful systems in HOL ?

- Use Monads !!!
  - The transition in an automaton $(\sigma,(\iota \times o),\sigma)set$ can isomorphically represented by:

$$\iota \Rightarrow (o \times \sigma)\ Mon_{SBE}$$

or for a deterministic transition function:

$$\iota \Rightarrow (o \times \sigma)\ Mon_{SE}$$

… which category theorists or functional programmers would recognize as a Monad function space

How to model and test stateful systems in HOL ?

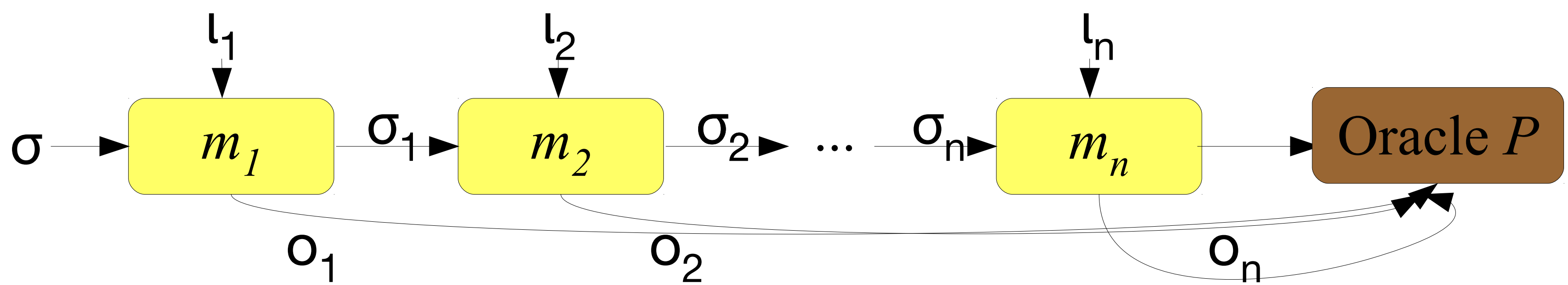
- Monads must have two combination operations bind and unit enjoying three algebraic laws.
  - For the concrete case of Mon_{SE}:

```
definition bindSE :: "('o,'σ)MONSE ⇒('o ⇒('o','σ)MONSE) ⇒('o','σ)MONSE"
where      "bindSE f g = (λσ. case f σof None ⇒None
                                   | Some (out, σ') ⇒g out σ')"

definition unitSE :: "'o ⇒('o, 'σ)MONSE" ("(return _)" 8)
where      "unitSE e = (λσ. Some(e,σ))"
```

  - and write $o \leftarrow m;\ m'\ o$ for $bind_{SE}\ m\ (\lambda o.\ m'\ o)$
    and $return$ for $unit_{SE}$

How to model and test stateful systems in HOL ?

- Valid Test Sequences: (_ ⊨ _)



- … are computable iff m_i are computable and the oracle P is true

- … can be symbolically executed …

$$\frac{}{(\sigma \models \text{return } P) = P}$$

$$\frac{C_m\, \iota\, \sigma \qquad m\, \iota\, \sigma = None}{(\sigma \models ((s \leftarrow m\, \iota; m'\; s))) = False}$$

$$\frac{C_m\, \iota\, \sigma \qquad m\, \iota\, \sigma = Some(b, \sigma')}{(\sigma \models s \leftarrow m\, \iota; m'\; s) = (\sigma' \leftarrow (m'\; b))}$$

How to model and test stateful systems in HOL ?

- Valid Test Sequences:

$$\sigma \models o_1 \leftarrow m_1\ \iota_1; \ldots ; o_n \leftarrow m_n\ \iota_n; \mathrm{return}(P\ o_1 \cdots o_n)$$

- … can be generated to code
- … can be symbolically executed …

$$\frac{}{(\sigma \models \mathrm{return}\ P) = P}$$

$$\frac{C_m\ \iota\ \sigma \qquad m\ \iota\ \sigma = None}{(\sigma \models ((s \leftarrow m\ \iota; m'\ s))) = False}$$

$$\frac{C_m\ \iota\ \sigma \qquad m\ \iota\ \sigma = Some(b, \sigma')}{(\sigma \models s \leftarrow m\ \iota; m'\ s) = (\sigma' \leftarrow (m'\ b))}$$

Conclusion

Monads offer a framework for symbolic computation

By embedding conditionals and loops, they can be used to white-box tests of programs …

… in a formally proven setting

Conclusion

Monadic approach to sequence testing:

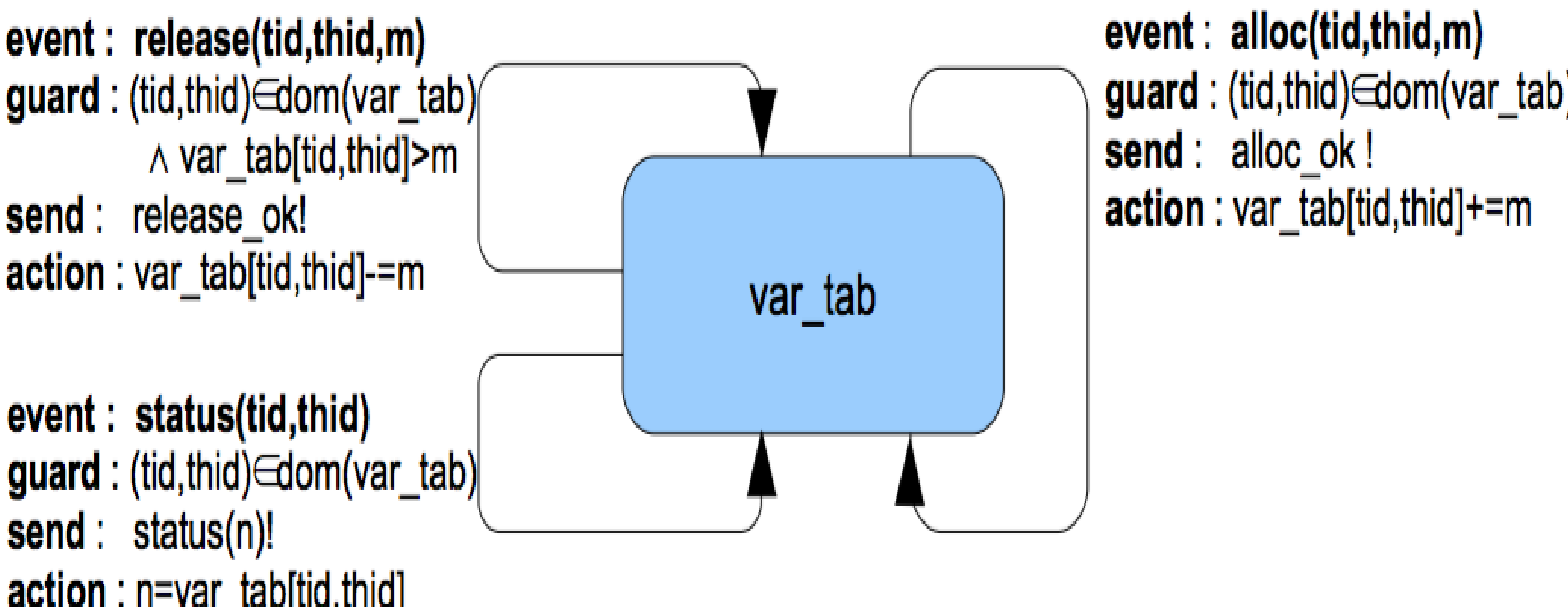
1. no surrender to finitism and constructivism
2. sensible shift from syntax to semantics: computations + compositions, not nodes + arcs
3. explicit difference between input and output,
4. theoretical and practical framework of numerous conformance notions,
5. new ways to new calculi of symbolic evaluation

Example : MyKeOS ?

- We consider an (brutal) abstraction of an L4 Kernel IPC protocol called “MyKeOS”
- It has
  - unbounded number of tasks
  - … having an unbounded number of threads
  - … which each have a counter for a resource
  - … the atomic actions alloc, release, status (tagged by task-id, thread-id, arguments)
  - release can only release allocated ressources

Example : MyKeOS ?

- A Semi-Formalization as ESFM



Example : MyKeOS ?

- State : $(task\_id \times thread\_id) \rightarrow int$
- Input events:

$$
\begin{aligned}
in_{event} = & \ alloc \quad task\_id\ thread\_id\ nat \\
& |\ release\ task\_id\ thread\_id\ nat \\
& |\ status \quad task\_id\ thread\_id
\end{aligned}
$$

- Output events:

$$out_{event} = alloc\_ok \ |\ release\_ok \ |\ status\_ok\ nat$$

- System Model SYS: interprets input event in a state and yields an output event and a successor state if successful, an exception otherwise.

Example : MyKeOS (0)

$\sigma_0 \models s \leftarrow$ mbind [alloc tid 1 m'',
release tid 0 m',
release tid 1 m''',
status tid 1] SYS;
unit(x = s)

Example : MyKeOS (0)

$\sigma_0 \models s \leftarrow$ mbind [alloc tid 1 m'',
release tid 0 m',
release tid 1 m''',
status tid 1] SYS;
unit(x = s)

Example : MyKeOS (2)

$(tid, 1) \in dom\ \sigma_0 \Longrightarrow$
$\sigma'_0 = \sigma_0((tid, 1) \mapsto the\ (\sigma_0\ (tid, 1)) + int\ m'') \Longrightarrow$

$\sigma'_0 \models s \leftarrow mbind\ [release\ tid\ 0\ m',$
$release\ tid\ 1\ m''',$
$status\ tid\ 1]\ SYS;$
$unit(x = alloc\_ok\ \#\ s)$

Example : MyKeOS (2)

$(tid, 1) \in dom\ \sigma_0 \Longrightarrow$

$\sigma'_0 = \sigma_0((tid, 1) \mapsto the\ (\sigma_0\ (tid, 1)) + int\ m'') \Longrightarrow$

$int\ m' \leq the\ ((\sigma_0((tid,1) \mapsto the(\sigma_0(tid,1))+int\ m''))$
$(tid,0)) \Longrightarrow$

$\sigma''_0 = \sigma'((tid, 0) \mapsto the\ (\sigma'(tid, 0))\ -\ int\ m') \Longrightarrow$

$\sigma''_0 \models s \leftarrow mbind\ [release\ tid\ 1\ m''',$
$status\ tid\ 1]\ SYS;$
$unit(x = alloc\_ok\ \#\ release\_ok\ \#\ s)$

Example : MyKeOS (3)

$(tid, 1) \in dom\ \sigma_0 \Longrightarrow$

$\sigma'_0 = \sigma_0((tid, 1) \mapsto the\ (\sigma_0\ (tid, 1)) + int\ m'') \Longrightarrow$

$int\ m' \leq the\ ((\sigma_0((tid,1) \mapsto the(\sigma_0(tid,1))+int\ m''))(tid,0)) \Longrightarrow$

$\sigma''_0 = \sigma'((tid, 0) \mapsto the\ (\sigma'(tid, 0)) - int\ m') \Longrightarrow$

$\sigma''_0 \models s \leftarrow mbind\ [release\ tid\ 1\ m''',$

$status\ tid\ 1]\ SYS;$

$unit(x = alloc\_ok\ \#\ release\_ok\ \#\ s)$

Example : MyKeOS (3)

$(tid, 1) \in \mathrm{dom}\ \sigma_0 \Longrightarrow$

$\sigma'_0 = \sigma_0((tid, 1) \mapsto \mathrm{the}\ (\sigma_0\ (tid, 1)) + \mathrm{int}\ m'') \Longrightarrow$

$\mathrm{int}\ m' \leq \mathrm{the}\ ((\sigma_0((tid,1)\mapsto \mathrm{the}(\sigma_0(tid,1))+\mathrm{int}\ m''))(tid,0)) \Longrightarrow$

$\sigma''_0 = \sigma'((tid, 0) \mapsto \mathrm{the}\ (\sigma'(tid, 0)) - \mathrm{int}\ m') \Longrightarrow$

$\ldots \Longrightarrow \ldots \Longrightarrow$

$\sigma'''_0 \vDash s \leftarrow$ mbind [status tid 1] SYS;
unit(x = alloc_ok # release_ok # release_ok # s)

Example : MyKeOS (4)

$(tid, 1) \in \text{dom } \sigma_0 \Longrightarrow$

$\sigma'_0 = \sigma_0((tid, 1) \mapsto \text{the } (\sigma_0\ (tid, 1)) + \text{int } m'') \Longrightarrow$

$\text{int } m' \leq \text{the } ((\sigma_0((tid,1) \mapsto \text{the}(\sigma_0(tid,1)) + \text{int } m''))(tid,0)) \Longrightarrow$

$\sigma''_0 = \sigma'((tid, 0) \mapsto \text{the } (\sigma'(tid, 0)) - \text{int } m') \Longrightarrow$

$\ldots \Longrightarrow \ldots \Longrightarrow$

$\sigma'''_0 \models$ s ← mbind [status tid 1] SYS;

unit(x = alloc_ok # release_ok # release_ok # s)

Example : MyKeOS (5)

$(tid, 1) \in \text{dom } \sigma_0 \Longrightarrow$

$\sigma'_0 = \sigma_0((tid, 1) \mapsto \text{the } (\sigma_0 \text{ } (tid, 1)) + \text{int } m'') \Longrightarrow$

$\text{int } m' \leq \text{the } ((\sigma_0((tid,1) \mapsto \text{the}(\sigma_0(tid,1))+\text{int } m''))(tid,0)) \Longrightarrow$

$\sigma''_0 = \sigma'((tid, 0) \mapsto \text{the } (\sigma'(tid, 0)) - \text{int } m') \Longrightarrow$

$\ldots \Longrightarrow \ldots \Longrightarrow \ldots \Longrightarrow \ldots \Longrightarrow$

$\sigma'''_0 \models s \leftarrow \text{mbind [] SYS;}$

unit(x = alloc_ok # release_ok # release_ok #
status_ok (the(σ'''_0 (tid,1))) # s)

Example : MyKeOS (6)

$(tid, 1) \in \text{dom}\ \sigma_0 \Longrightarrow$
$\sigma'_0 = \sigma_0((tid, 1) \mapsto \text{the}\ (\sigma_0\ (tid, 1)) + \text{int}\ m'') \Longrightarrow$
$\text{int}\ m' \leq \text{the}\ ((\sigma_0((tid,1) \mapsto \text{the}(\sigma_0(tid,1)) + \text{int}\ m''))(tid,0)) \Longrightarrow$
$\sigma''_0 = \sigma'((tid, 0) \mapsto \text{the}\ (\sigma'(tid, 0)) - \text{int}\ m') \Longrightarrow$
$\ldots \Longrightarrow \ldots \Longrightarrow \ldots \Longrightarrow \ldots \Longrightarrow$

$\sigma'''_0 \models s \leftarrow \text{mbind}\ []\ \text{SYS};$
unit(x = alloc_ok # release_ok # release_ok #
status_ok (the(σ'''_0 (tid,1))) # s)

Example : MyKeOS (6)

$(tid, 1) \in dom\ \sigma_0 \Longrightarrow$
$\sigma'_0 = \sigma_0((tid, 1) \mapsto the\ (\sigma_0\ (tid, 1)) + int\ m'') \Longrightarrow$
$int\ m' \leq the\ ((\sigma_0((tid,1) \mapsto the(\sigma_0(tid,1)) + int\ m''))(tid,0)) \Longrightarrow$
$\sigma''_0 = \sigma'((tid, 0) \mapsto the\ (\sigma'(tid, 0)) - int\ m') \Longrightarrow$
$... \Longrightarrow ... \Longrightarrow ... \Longrightarrow ... \Longrightarrow$

$\sigma'''_0 \models unit(x = [alloc\_ok, release\_ok, release\_ok, status\_ok\ (the(\sigma'''_0\ (tid,1)))])$

Example : MyKeOS (7)

$(tid, 1) \in dom\ \sigma_0 \Longrightarrow$
$\sigma'_0 = \sigma_0((tid, 1) \mapsto the\ (\sigma_0\ (tid, 1)) + int\ m'') \Longrightarrow$
$int\ m' \leq the\ ((\sigma_0((tid,1) \mapsto the(\sigma_0(tid,1))+int\ m''))(tid,0)) \Longrightarrow$
$\sigma''_0 = \sigma'((tid, 0) \mapsto the\ (\sigma'(tid, 0)) - int\ m') \Longrightarrow$
$... \Longrightarrow ... \Longrightarrow ... \Longrightarrow ... \Longrightarrow$

x = [alloc_ok, release_ok, release_ok, status_ok (the(σ'''_0 (tid,1)))])

How to model and test stateful systems ?

- Test Refinements for a step-function SPEC and a step function SUT:

$$\sigma \models o_1 \leftarrow \mathrm{SPEC}_1\ \iota_1; \ldots; o_n \leftarrow \mathrm{SPEC}_n\ \iota_n; \mathrm{return}(res = [o_1 \cdots o_n])$$

$$\longrightarrow$$

$$\sigma \models o_1 \leftarrow \mathrm{SUT}_1\ \iota_1; \ldots; o_n \leftarrow \mathrm{SUT}_n\ \iota_n; \mathrm{return}(res = [o_1 \cdots o_n])$$

- The premisse is reduced by symbolic execution to constraints over *res*; a constraint solver (Z3) produces an instance for *res*. The conclusion is compiled to a test-driver/test-oracle linked to *SUT*.

Explicit Test-Refinements

- This motivates the notion of a "Generalized Monadic Test-Refinement"

$$(I \sqsubseteq_{\langle \Sigma_0, CC, conf \rangle} S) =$$

$$(\forall\ \sigma_0 \in \Sigma_0.\ \ \forall\ \ \iota s \in CC.\ \forall\ res.$$

$$(\sigma_0 \Box (os \leftarrow mbind\ \iota s\ S;\ return\ (conf\ \iota s\ os\ res)))$$

$$\longrightarrow$$

$$(\sigma_0 \Box (os \leftarrow mbind\ \iota s\ I;\ return\ (conf\ \iota s\ os\ res))))$$

Explicit Test-Refinements (Inclusion)

- This motivates the notion of a "Generalized Monadic Test-Refinement"

  With conf set to:

  - (λ is os x. length is = length os ∧ os=x)
    ==> Inclusion Test

$$I \sqsubseteq_{IS\langle\Sigma_0,CC\rangle} S$$

Explicit Test-Refinements (Deadlock)

- This motivates the notion of a "Generalized Monadic Test-Refinement"

  With conf set to:

  – (λ is os x. length is > length os $\wedge$ os=x)
    ==> Deadlock Refinement

$$I \sqsubseteq_{DR\langle \Sigma_0, CC \rangle} S$$

Explicit Test-Refinements (IOCO)

- This motivates the notion of a "Generalized Monadic Test-Refinement"

  With conf set to:

  - (λ is os x. length is = length os $\wedge$ post_cond (last os) $\wedge$ os=x)

    ==> IOCO Refinement (without quiescense)

$$I \sqsubseteq_{IOCO\langle \Sigma_0, CC \rangle} S$$

Some Theory on Test-Refinements

- This motivates the notion of a "Generalized Monadic Test-Refinement"

  One can now PROVE equivalences between different members of the test-refinement families

  … and prove alternative forms for efficiency optimizations of the generated test-driver code.

Some Theory on Test-Refinements

- This motivates the notion of a
  "Generalized Monadic Test-Refinement"

  One can now PROVE equivalences between
  different members of the test-refinement families

  … and prove alternative forms for efficiency
  optimizations of the generated test-driver code.

Some Theory on Test-Refinements

- For example:

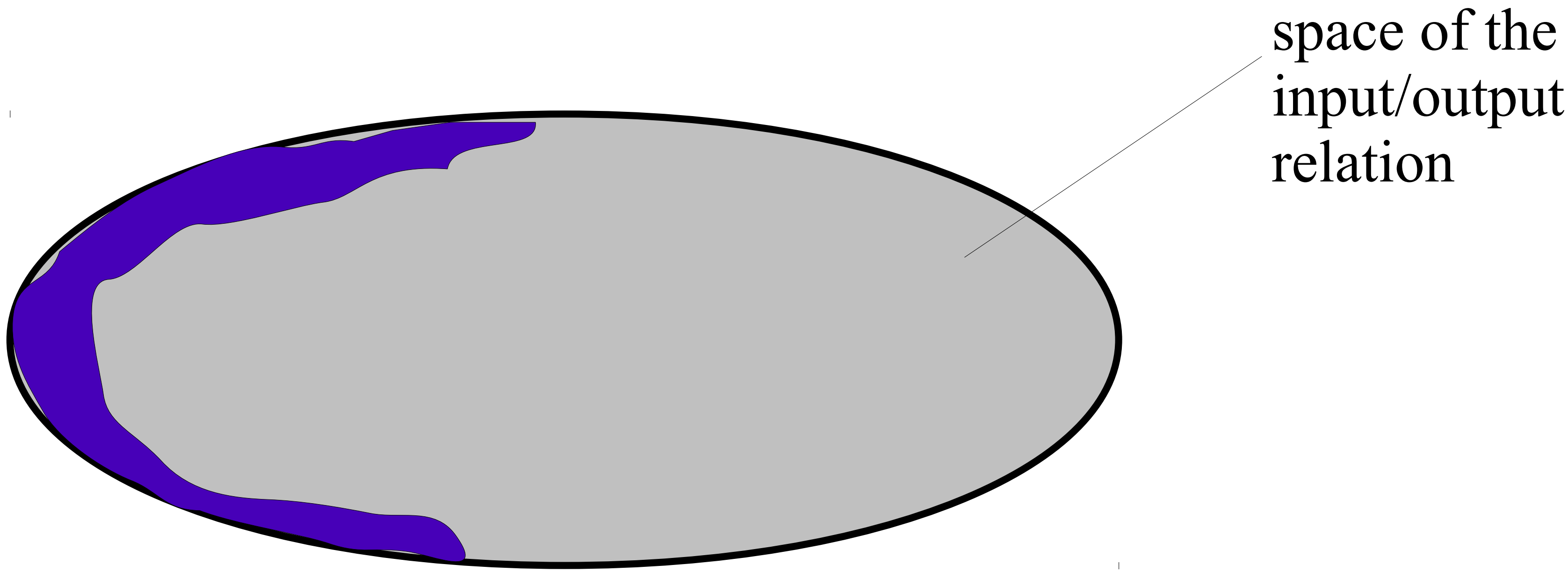
$$\frac{\begin{array}{c}\left[\begin{array}{l}\sigma_0 \in Init, \iota s \in CC, \\ \sigma_0 \models os \leftarrow \mathrm{mbind}_{\mathrm{FailStop}}\ \iota s\ S; \mathrm{unit}_{\mathrm{SE}}(os = res)\end{array}\right]_{\sigma_0\ \iota s\ res} \\ \vdots \\ \sigma_0 \models os \leftarrow \mathrm{mbind}_{\mathrm{FailStop}}\ \iota s\ I; \mathrm{unit}_{\mathrm{SE}}(os = res)\end{array}}{I \sqsubseteq_{IT\langle Init, CC\rangle} S}$$

- For example:

theorem ioco_VS_IOCO:
  assumes "strictly_IO_alternating S" and "io_deterministic S"
  shows "∃ S'. I ioco S = ((two_step I) $\sqsubseteq_{\mathrm{IOCO}\langle\{x.True\},\{x.True\}\rangle}$ S'"

Alternatives in Testgeneration

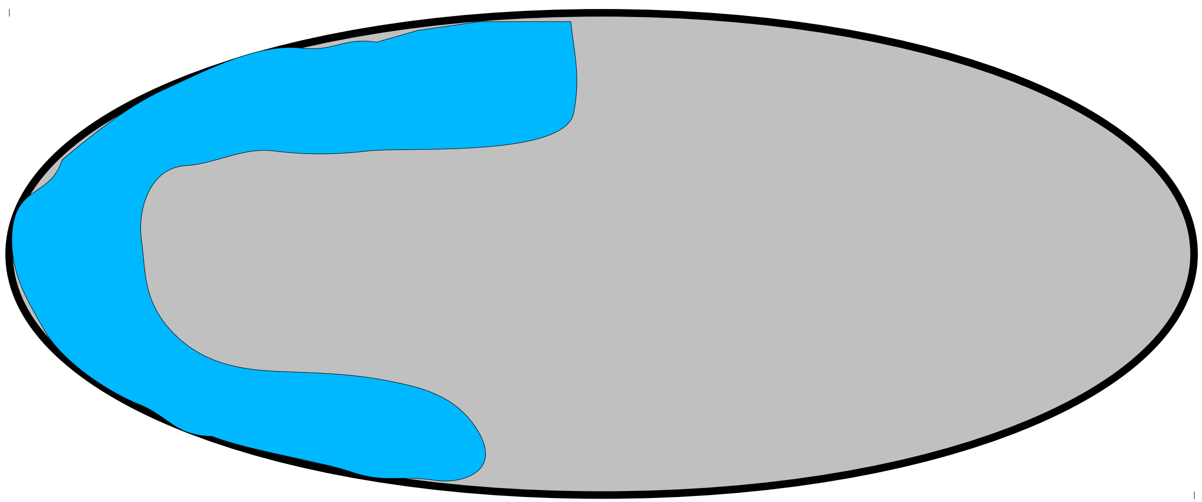
- Counter-example generation based on finite sub-model generation and SAT-solving (nitpick, kodkod, and co)



model-depth 10

Alternatives in Testgeneration

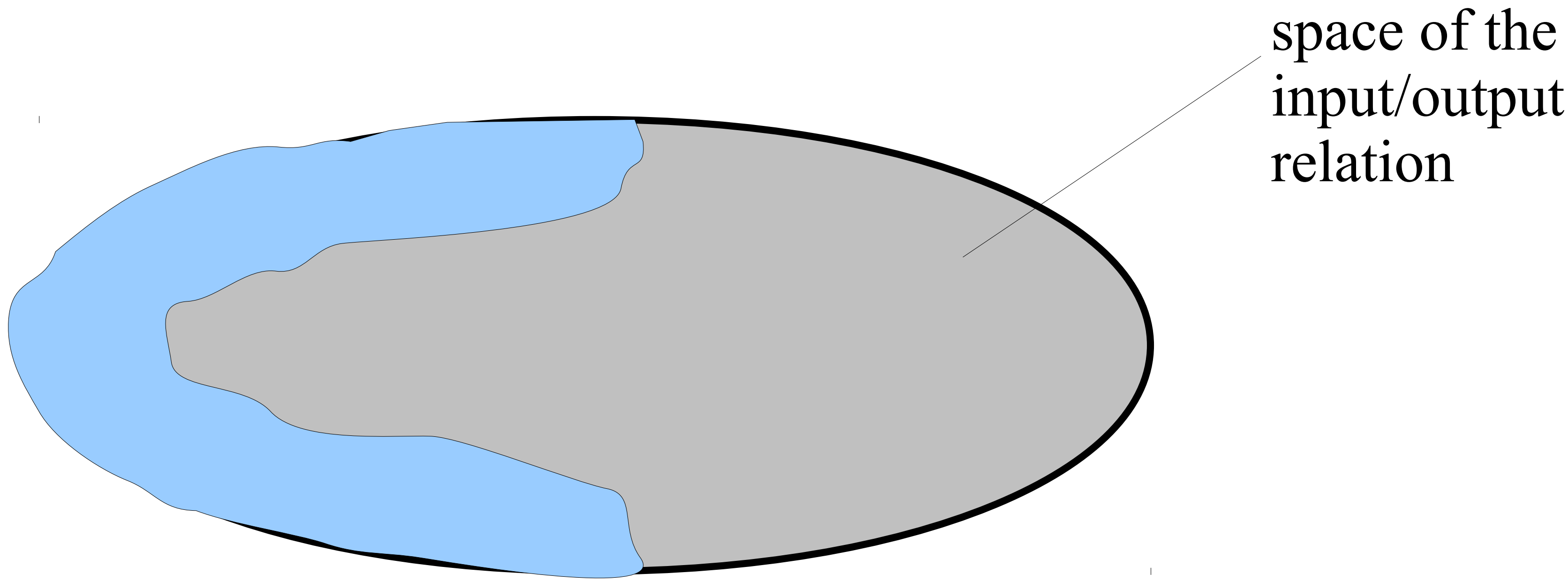
- Counter-example generation based on finite sub-model generation and SAT-solving (nitpick, kodkod, and co)



model-depth 100

Alternatives in Testgeneration

- Counter-example generation based on finite sub-model generation and SAT-solving (nitpick, kodkod, and co)

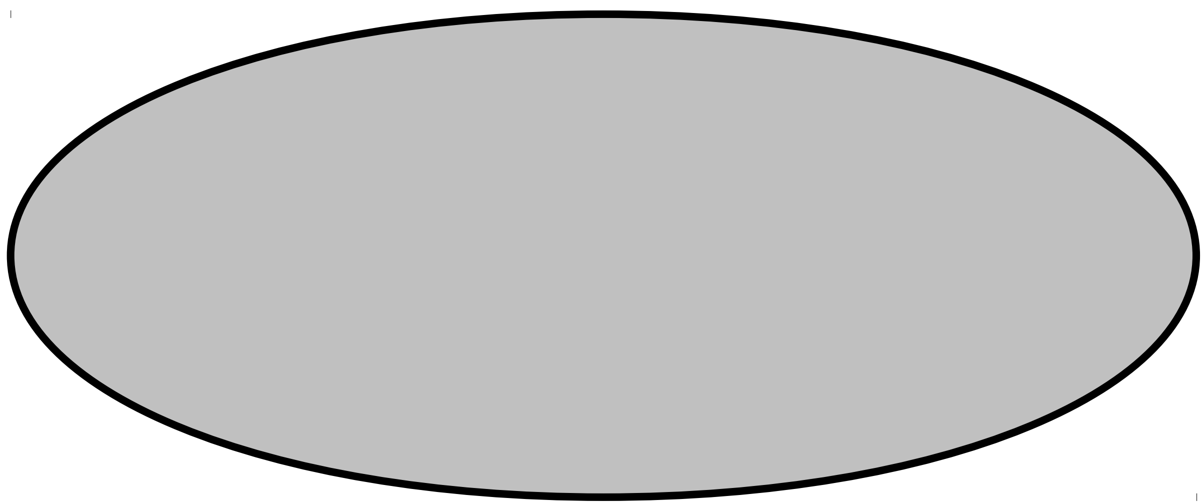


model-depth 10

bias towards small values,
impossibility to catch infinite models

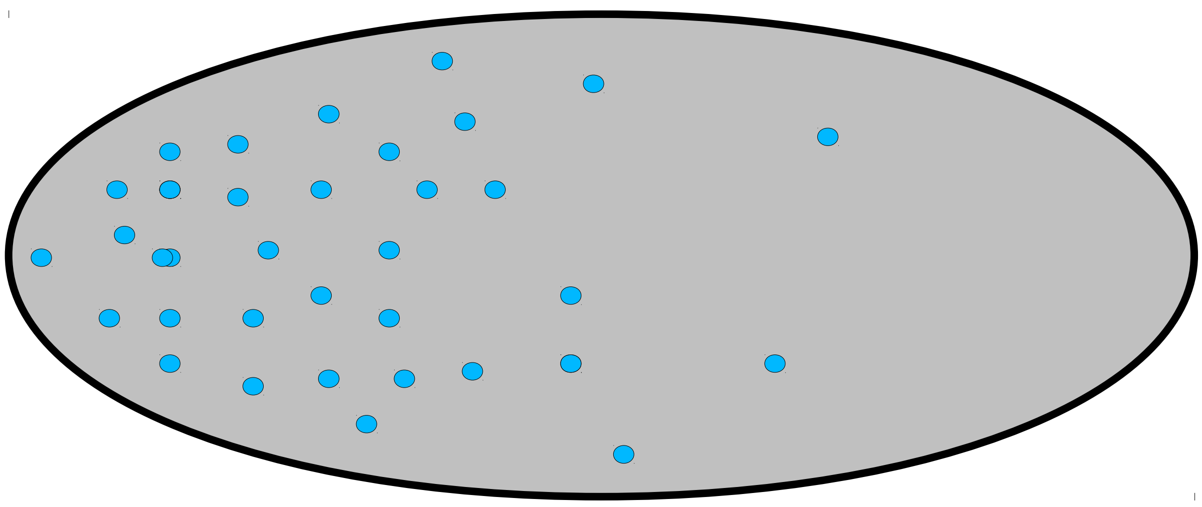
Alternatives in Testgeneration

- Random-testing a la Quickcheck



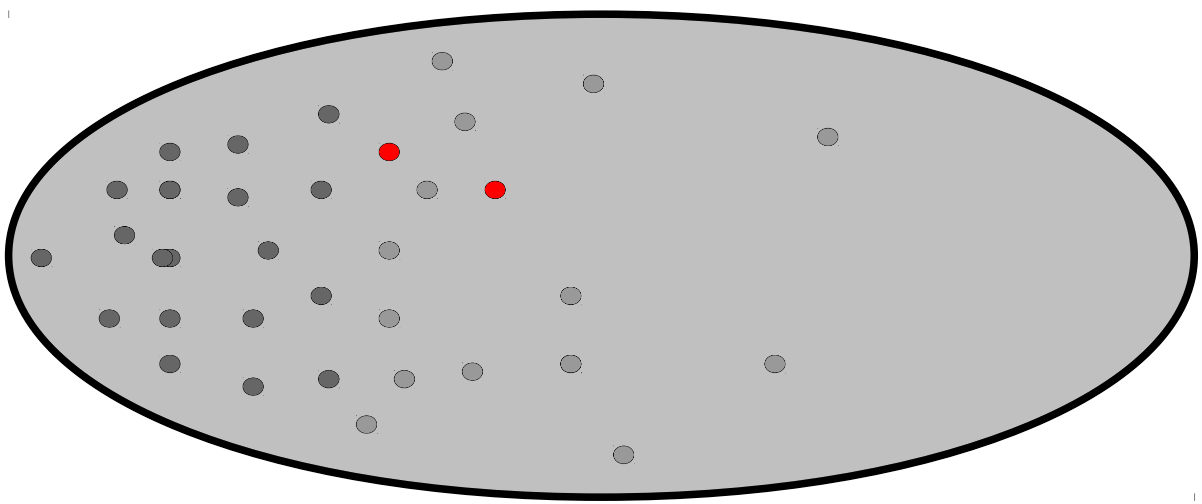
Alternatives in Testgeneration

- Random-testing a la Quickcheck



Alternatives in Testgeneration

- Random-testing a la Quickcheck



in complex, probability to find a feasible test-case are extremely low.
Leads to hand-programmed random-generators ...

Alternatives in Testgeneration

- Error-based Generation Methods

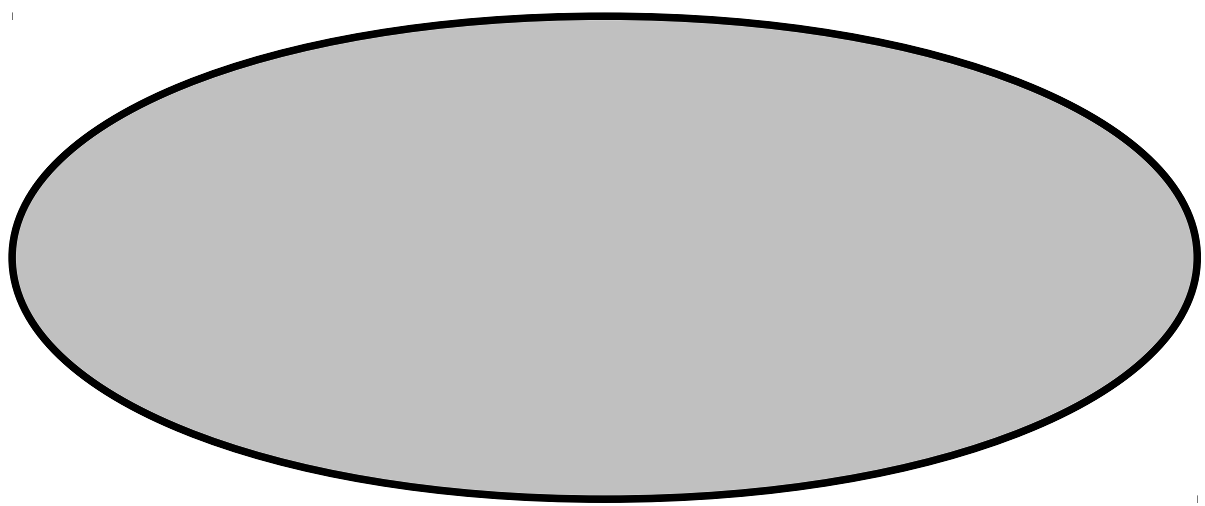
  "Mutant Testing"

  Depends crucially on the availability of Error-Models:

  - implementation-based : can make sense
  - specification-based : ???

Our Approach:

- DNF based case-splitting, normalization modulo E, test-data-selection via SAT or SMT solvers



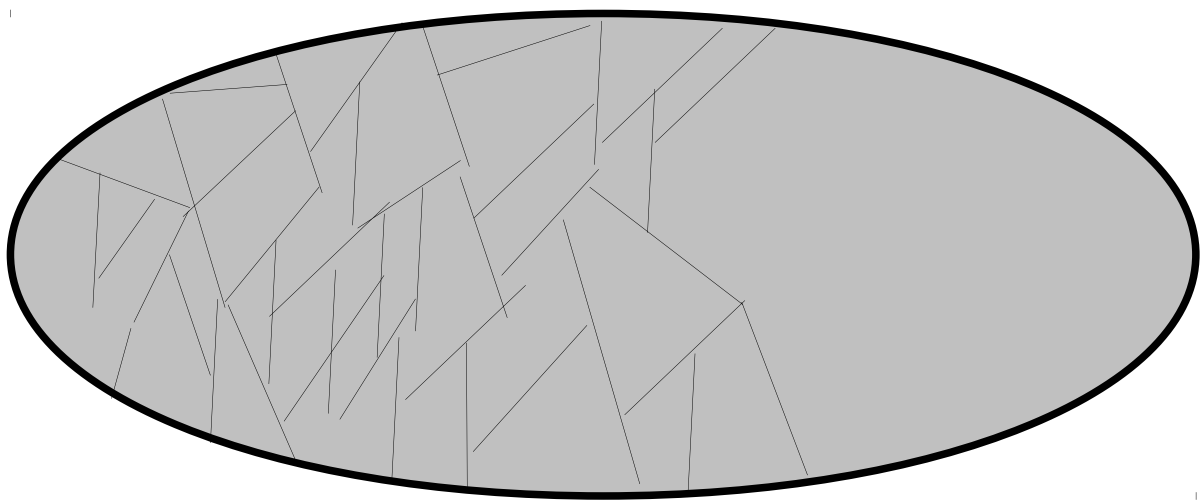
Example : MyKeOS (1)

$(tid, 1) \in dom\ \sigma_0 \Longrightarrow$
$\sigma'_0 = \sigma_0((tid, 1) \mapsto the\ (\sigma_0\ (tid, 1)) + int\ m'') \Longrightarrow$

$\sigma'_0 \models s \leftarrow$ mbind [release tid 0 m',
release tid 1 m''',
status tid 1] SYS;
unit(x = alloc_ok # s)

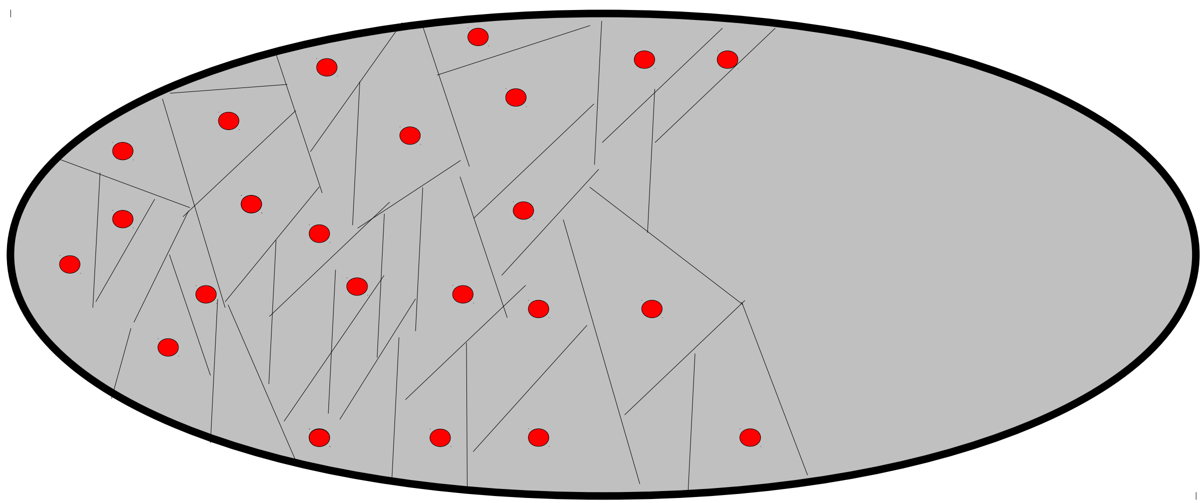
Our Approach:

- DNF based case-splitting, normalization modulo E, test-data-selection via SAT or SMT solvers



Our Approach:

- DNF based case-splitting, normalization modulo E, test-data-selection via SAT or SMT solvers

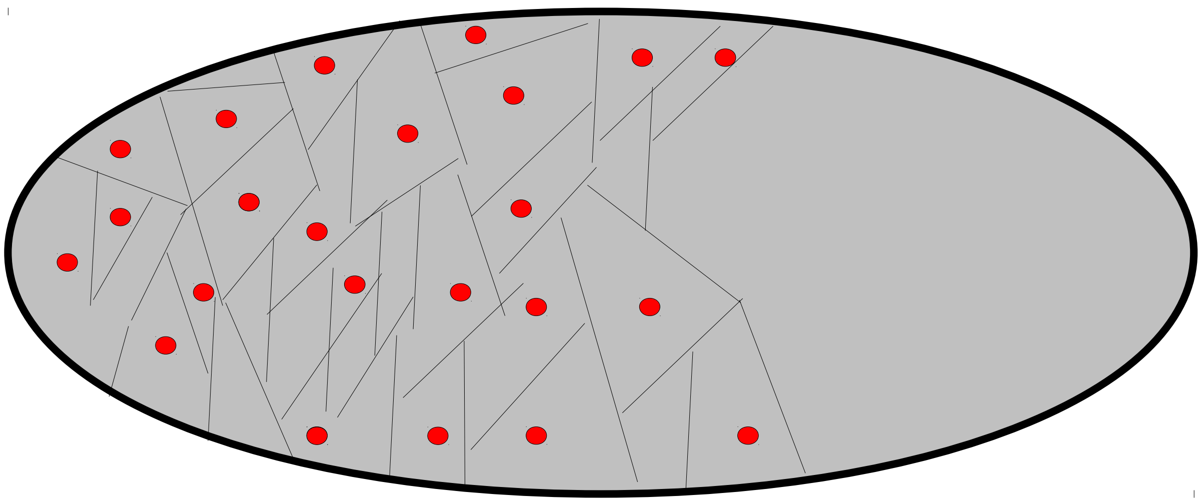


- Less bias, clear criterion DNF_E

- Can handle infinite data spaces via symbolic execution

Our Approach:

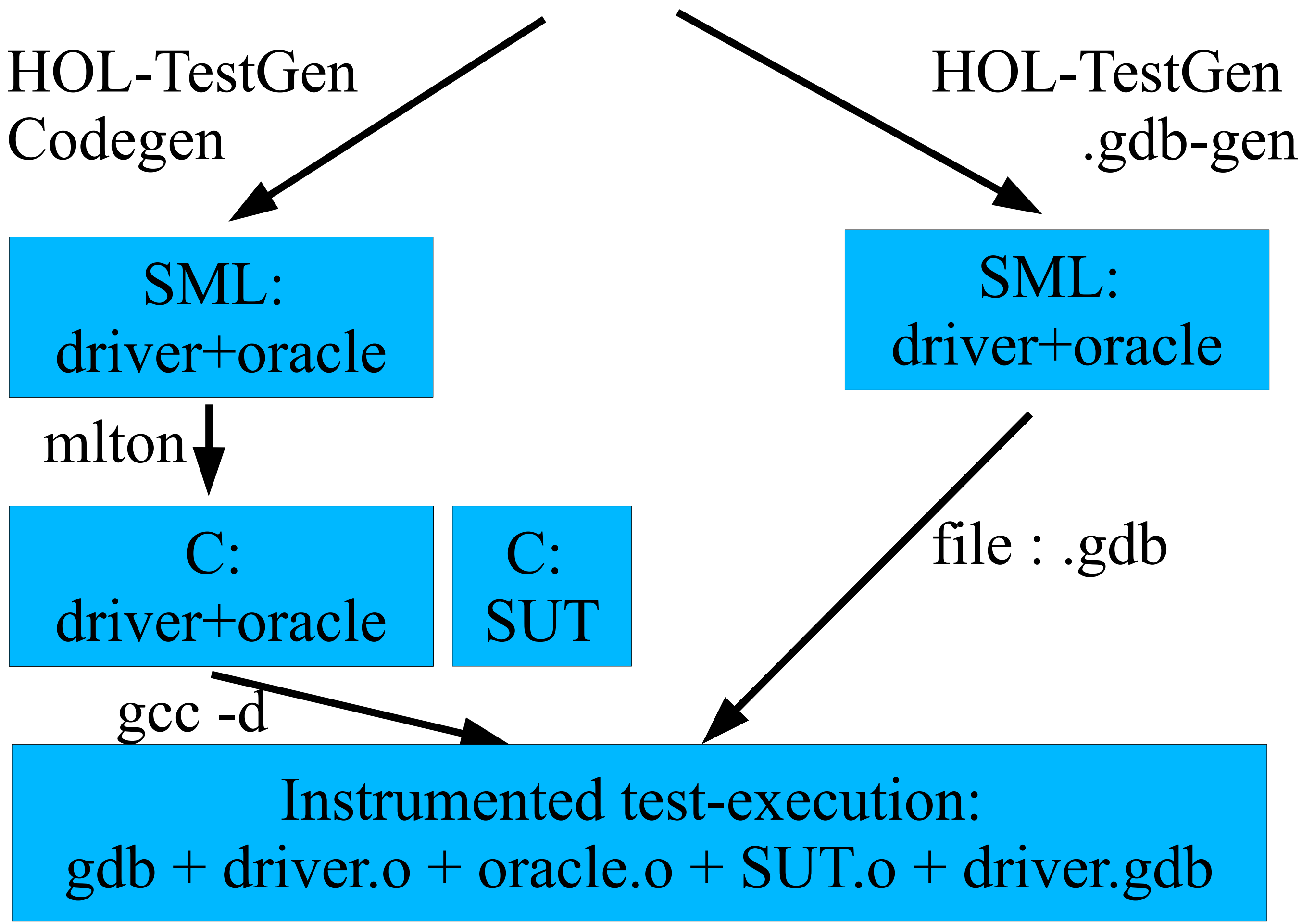
- DNF based case-splitting, normalization modulo E, test-data-selection via SAT or SMT solvers



- But why are so few systems that try to implement this for sequence testing???

Practice : How to test concurrent programs ?

$$\sigma \models o_1 \leftarrow \mathrm{SUT}_1\ \iota_1; \ldots; o_n \leftarrow \mathrm{SUT}_n\ \iota_n; \mathrm{return}(res = [o_1 \cdots o_n])$$



Practice : How to test concurrent programs ?

- Assumption: Code compiled for LINUX and instrumented for debugging (gcc -d)
- Assumption: No dynamic thread creation (realistic for our target OS); identifiable atomic actions in the code;
- Assumption: Mapping from abstract atomic actions in the model to code-positions known.
- Abstract execution sequences were generated to .gdb scripts forcing explicit thread-switches of the SUT executed under gdb.

Practice : How to test concurrent programs ?

```
thread IP4_send(tid_rec, thid_rec){
        if (defined(tid_rec) &&
           defined(thid_rec)) {
                ...
                grab_lock();

               [atom: IPC_sendinit]
                ...
                if(curr_tid_hasRWin_tid_rec){
                        ...
                        grab_lock();

                       [atom: IPC_prep]
                        . . .
                        . . .
                }
                else{ return(ERROR_22);}
        }
        else{ return(ERROR_35);}
}
```

```
thread IP4_receive(tid_snd, thid_snd){
        if (defined(tid_snd) &&
           defined(thid_snd)) {
                ...
                grab_lock();

                re[atom: IPC_rec_rdy]
                ...
                if(curr_tid_hasRin_tid_rec)  {
                        ...
                        grab_lock();

                        re[atom: IPC_wait]
                        . . .
                        . . .
                }
                else{ return(ERROR_59);}
        }
        else{ return(ERROR_21);}
}
```

Practice : How to test concurrent programs ?

```
thread IP4_send(tid_rec, thid_rec){
        if (defined(tid_rec) &&
            defined(thid_rec)) {
                ...
                grab_lock();
"switch 2"
                atom: IPC_sendinit
                ...
                if(curr_tid_hasRWin_tid_rec){
                        ...
                        grab_lock();

                        atom: IPC_prep
                        . . .
                        . . .
                }
                else{ return(ERROR_22);}
        }
        else{ return(ERROR_35);}
}
```

```
thread IP4_receive(tid_snd, thid_snd){
"switch 1"
        if (defined(tid_snd) &&
            defined(thid_snd)) {
                ...
                grab_lock();

                re  atom: IPC_rec_rdy
"switch 1"
                ...
                if(curr_tid_hasRin_tid_rec)  {
                        ...
                        grab_lock();

                        re  atom: IPC_wait
"switch 1"
                        . . .
                        . . .
                }
                else{ return(ERROR_59);}
        }
        else{ return(ERROR_21);}
}
```

Practice : How to test concurrent programs ?

```
thread IP4_send(tid_rec, thid_rec){
    if (defined(tid_rec) &&
        defined(thid_rec)) {
            ...
            grab_lock();
"switch 2"
            atom: IPC_sendinit
            ...
            if(curr_tid_hasRWin_tid_rec){
                ...
                grab_lock();
                atom: IPC_prep
                . . .
                . . .
            }
            else{ return(ERROR_22);}
    }
    else{ return(ERROR_35);}
}
```

```
thread IP4_receive(tid_snd, thid_snd){
"switch 1"
    if (defined(tid_snd) &&
        defined(thid_snd)) {
            ...
            grab_lock();
            re  atom: IPC_rec_rdy
"switch 1"
            ...
            if(curr_tid_hasRin_tid_rec) {
                ...
                grab_lock();
                re  atom: IPC_wait
"switch 1"
                . . .
                . . .
            }
            else{ return(ERROR_59);}
    }
    else{ return(ERROR_21);}
}
```

Practice : How to test concurrent programs ?

- Computing the input sequence as interleaving of atomic actions of system-API-Calls:

$$[\iota_1,\ldots,\iota_n] \in \text{interleave}_\iota\ (\text{IPC\_send}\ t_2\ th_3)\ (\text{IPC\_receive}\ t_1\ th_7)$$

where ι_j is an input for an atomic action ...