Specification Format for Reactive Synthesis Problems

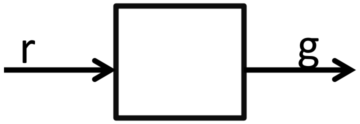
Ayrat Khalimov

SYNT 2015

TU Graz
Graz University of Technology

RiSE
Rigorous Systems Engineering

Simple arbiter



- “Every request should be granted”: $\mathbf{G}(r \ \rightarrow \mathbf{F}g)$
- “No spurious grants”

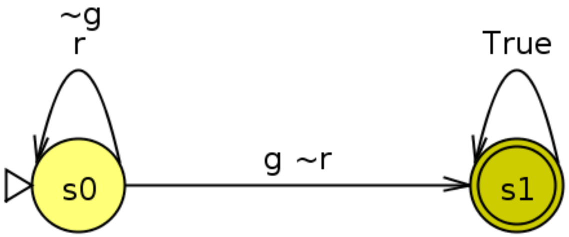
Let’s specify “spurious grants” in RE:

$$(.,.)^{*}(.,g)(\neg r,\neg g)^{+}\,(\neg r,g)$$

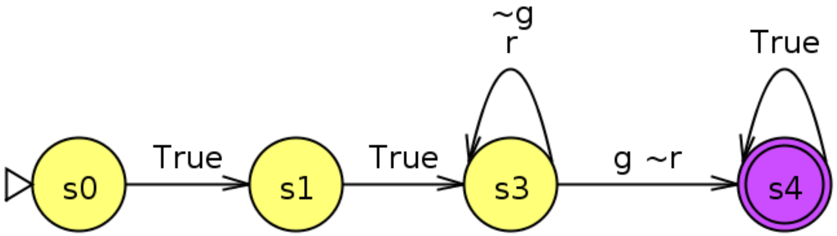
In LTL: $(.,.)^*(.,g)(\neg r,\neg g)^+(\neg r,g)$

- $\mathbf{F}(g\ \mathbf{U}\ \neg r\neg g\ \mathbf{U}\ \neg r\ g)$?

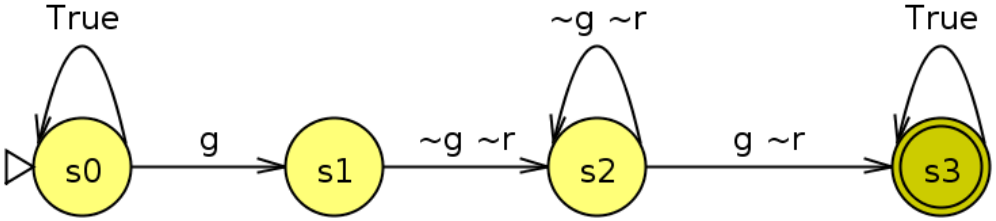
(NO! It accepts $(r\ \neg g)(\neg r\ g)$)



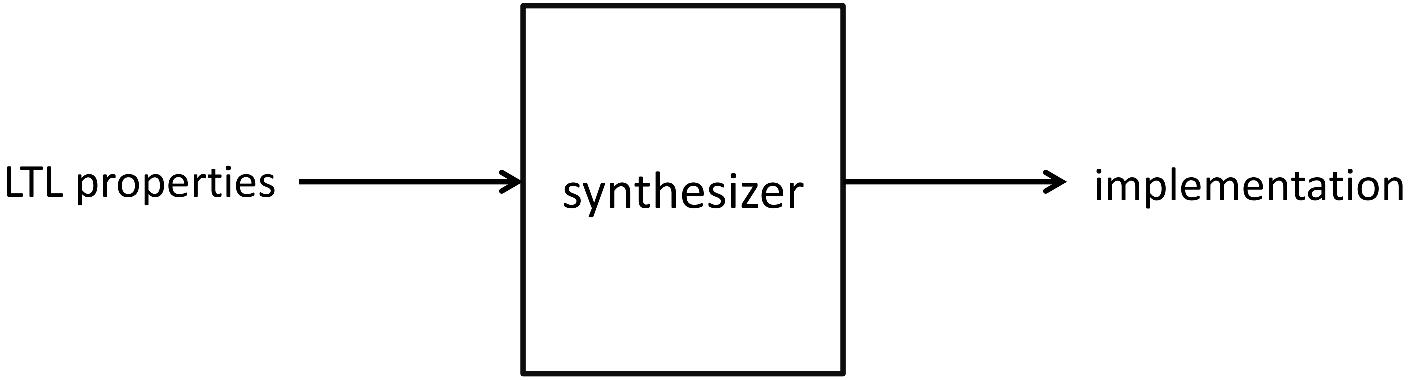
- $\mathbf{F}(g\ \mathbf{U}\ \mathbf{X}(\neg r\neg g\ \mathbf{U}\ \mathbf{X}\neg r\ g))$?



- $\mathbf{F}(g \wedge (g\ \mathbf{U}\ (\neg r\neg g \wedge (\neg r\neg g\ \mathbf{U}\ \neg r\ g))))$



Synthesis flow

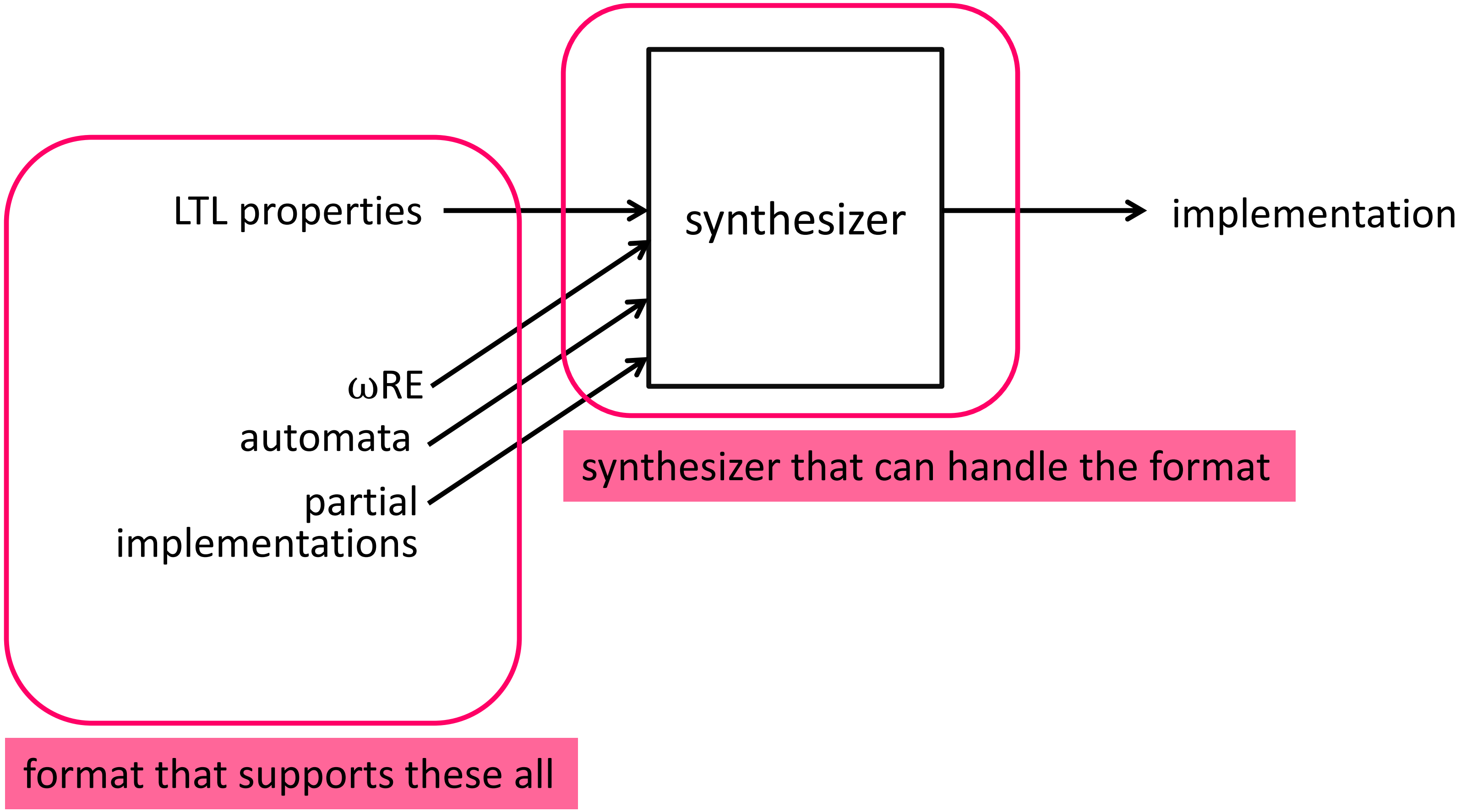


LTL properties

synthesizer

implementation

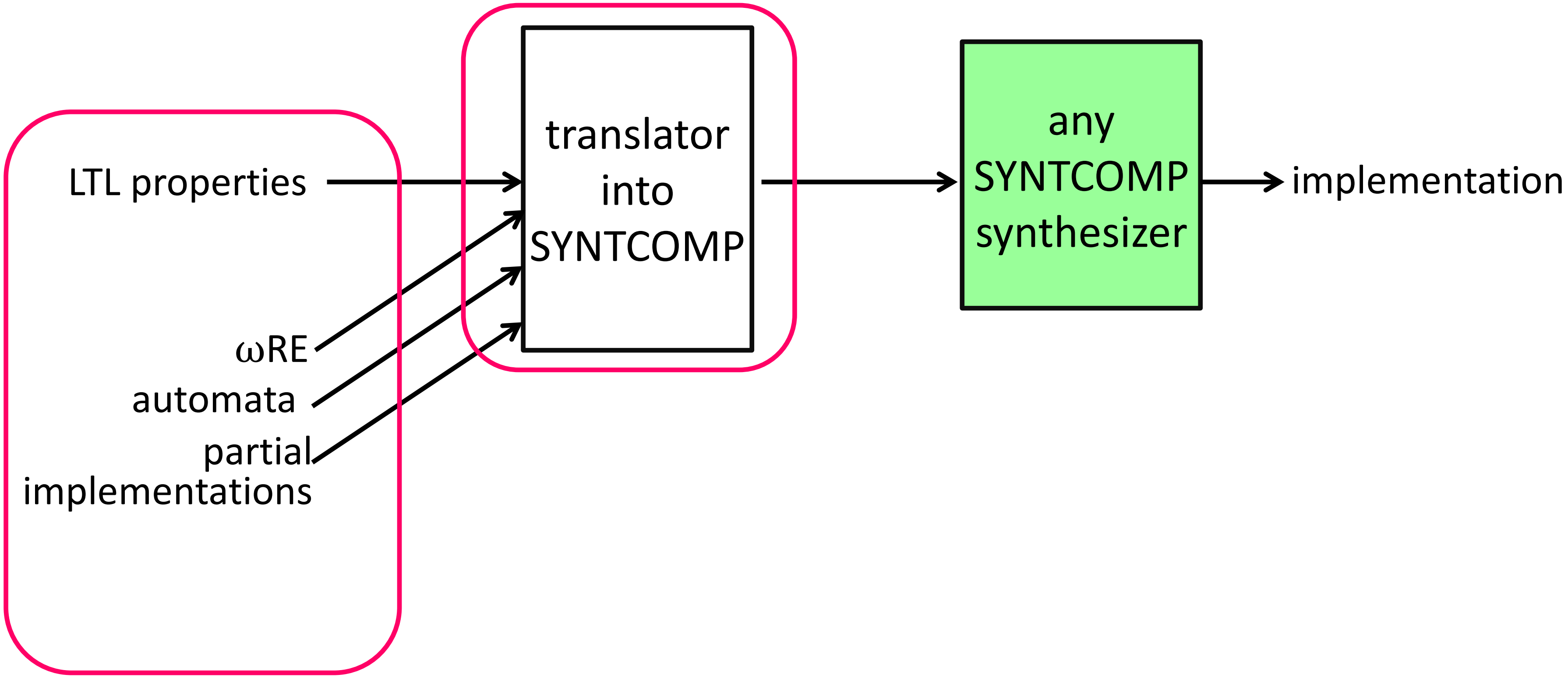
Synthesis flow



synthesizer that can handle the format

format that supports these all

Synthesis flow



LTL properties

ωRE

automata

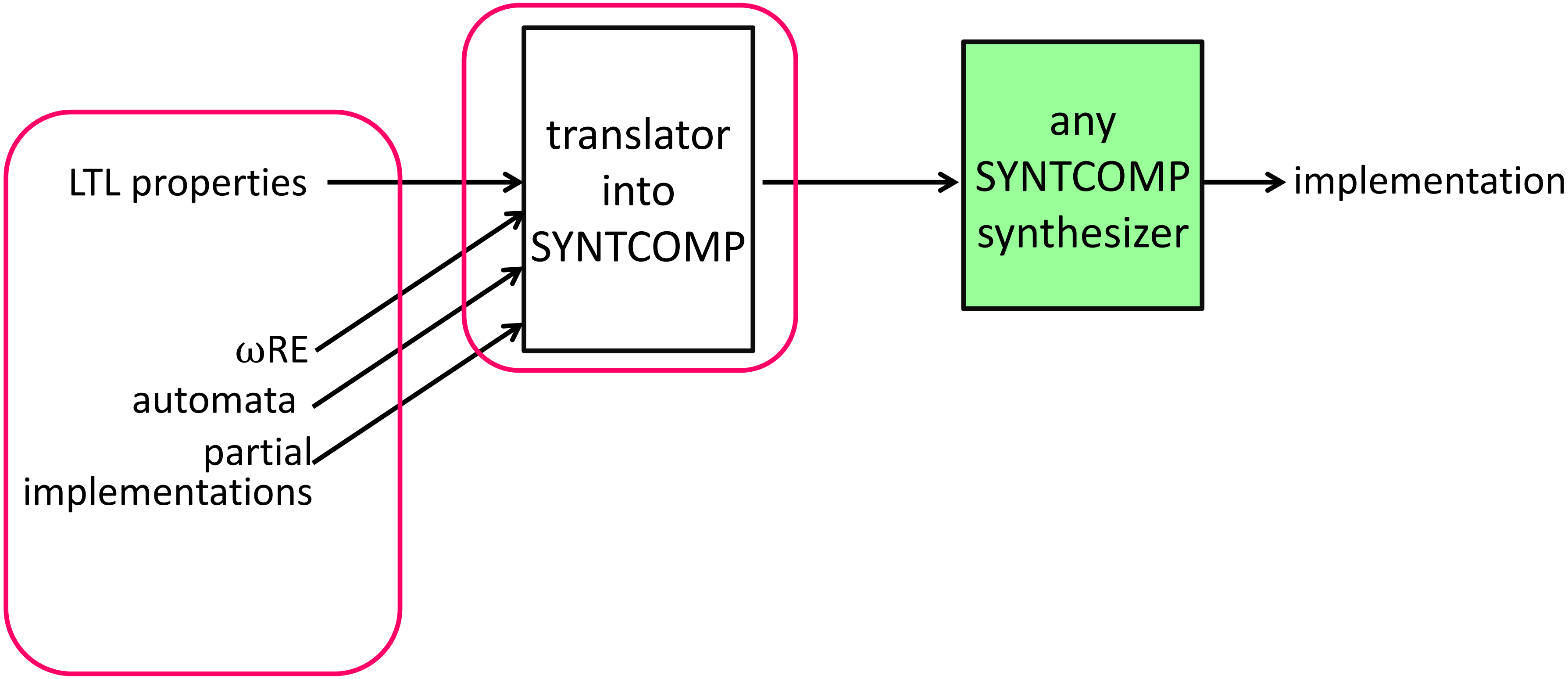
partial implementations

translator into SYNTCOMP

any SYNTCOMP synthesizer

implementation

Outline of the talk



new format (extended SMV)

translator extended SMV -> SYNTCOMP

synthesis example: a Huffman encoder

Format requirements

- embedded into existing programming language
- modular
- property language agnostic (LTL, ωRE, automata…)
- fast synthesizers

Proposed format

- embedded into existing programming language
  - SMV
- modular
  - part of SMV
- property language agnostic (LTL, ωRE, automata…)
  - automata
- fast synthesizers
  - SYNTCOMP

Comparison with ([1])([2])

- embedded into existing programming language
  - SMV *(SMV)* *(Promela)*
- modular
  - part of SMV *(part of SMV)* *(part of Promela)*
- property language agnostic (LTL, ωRE, automata…)
  - automata *(LTL patterns)* *(LTL + relations)*
- fast synthesizers
  - SYNTCOMP *(original GR1)* *(SLUGS GR1)*

FORMAT DESCRIPTION

EXTENDED SMV

SMV format

```
MODULE main
VAR
  input: 0..10;
  state: boolean;
  x: 0..10;
```

variables

```
DEFINE
  x_is_2input := (x=input+input);
```

macros

```
ASSIGN
  init(state) := FALSE;
  next(state) := (x=0 | x_is_2input);
  init(x) := 0;
  next(x) := x+input;
```

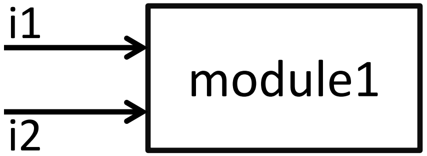
variables behaviour

```
LTLSPEC
  G(state | (x!=10))
```

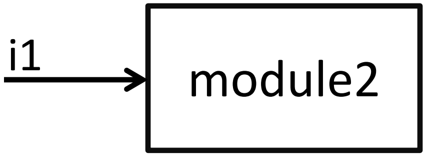
specification

SMV format (cont.)

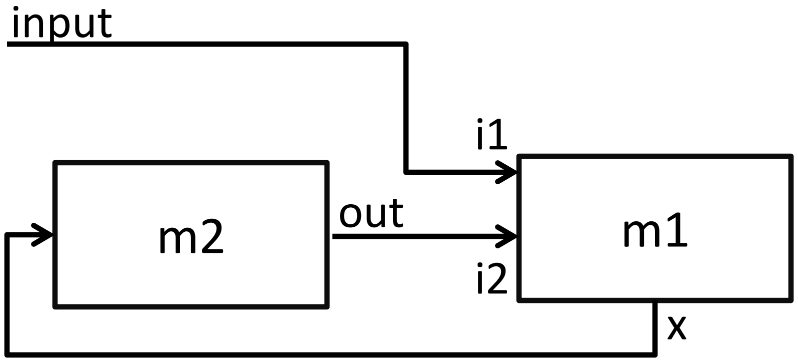
```
MODULE module1(i1,i2)
VAR
  x: ...
...
```



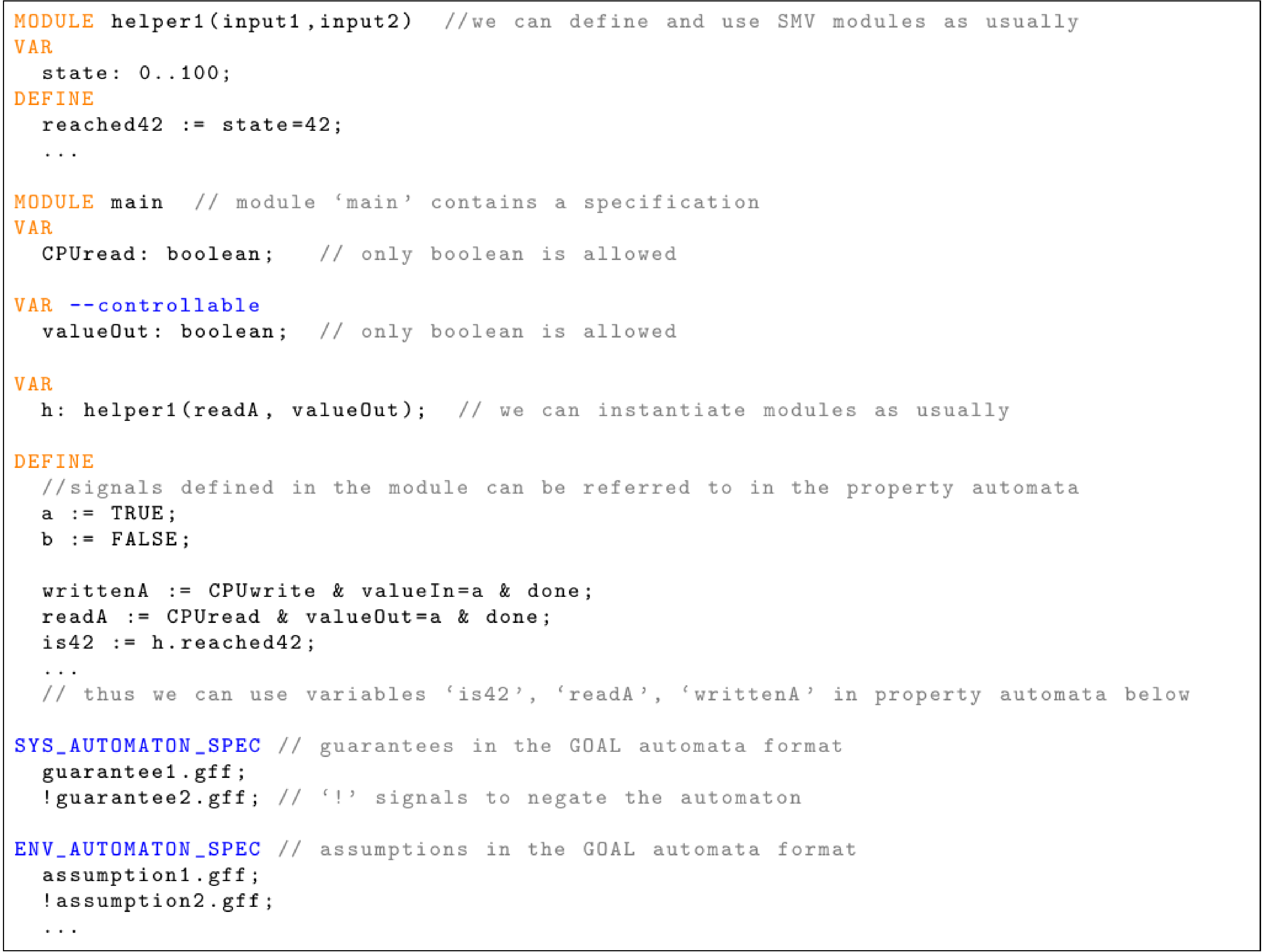
```
MODULE module2(i1)
VAR
  out : ...
```



```
MODULE main
VAR
  input: ...
VAR
  m1: module1(input, m2.out);
  m2: module2(m1.x);
```

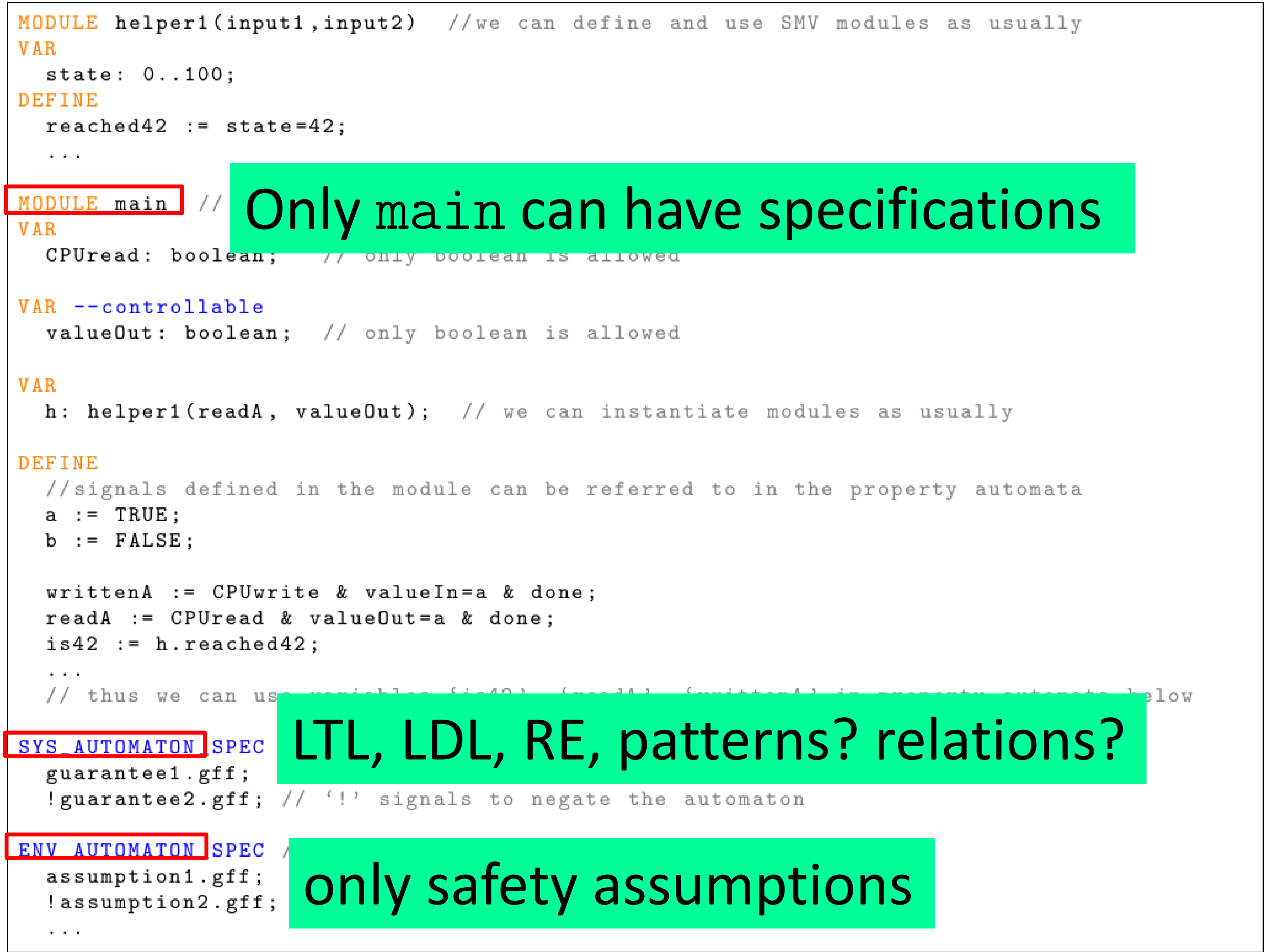


Extended SMV



```
MODULE helper1(input1,input2)  //we can define and use SMV modules as usually
VAR
  state: 0..100;
DEFINE
  reached42 := state=42;
  ...

MODULE main  // module ‘main’ contains a specification
VAR
  CPUread: boolean;    // only boolean is allowed

VAR --controllable
  valueOut: boolean;  // only boolean is allowed

VAR
  h: helper1(readA, valueOut);  // we can instantiate modules as usually

DEFINE
  //signals defined in the module can be referred to in the property automata
  a := TRUE;
  b := FALSE;

  writtenA := CPUwrite & valueIn=a & done;
  readA := CPUread & valueOut=a & done;
  is42 := h.reached42;
  ...
  // thus we can use variables ‘is42’, ‘readA’, ‘writtenA’ in property automata below

SYS_AUTOMATON_SPEC // guarantees in the GOAL automata format
  guarantee1.gff;
  !guarantee2.gff; // ‘!’ signals to negate the automaton

ENV_AUTOMATON_SPEC // assumptions in the GOAL automata format
  assumption1.gff;
  !assumption2.gff;
  ...
```

Extended SMV



```
MODULE helper1(input1,input2)  //we can define and use SMV modules as usually
VAR
  state: 0..100;
DEFINE
  reached42 := state=42;
  ...

MODULE main  //
VAR
  CPUread: boolean;   // only boolean is allowed

VAR --controllable
  valueOut: boolean;  // only boolean is allowed

VAR
  h: helper1(readA, valueOut);  // we can instantiate modules as usually

DEFINE
  //signals defined in the module can be referred to in the property automata
  a := TRUE;
  b := FALSE;

  writtenA := CPUwrite & valueIn=a & done;
  readA := CPUread & valueOut=a & done;
  is42 := h.reached42;
  ...
  // thus we can use variables {is42}, {readA}, {writtenA} in property automata below

SYS_AUTOMATON SPEC
  guarantee1.gff;
  !guarantee2.gff; // '!' signals to negate the automaton

ENV_AUTOMATON SPEC /
  assumption1.gff;
  !assumption2.gff;
  ...
```

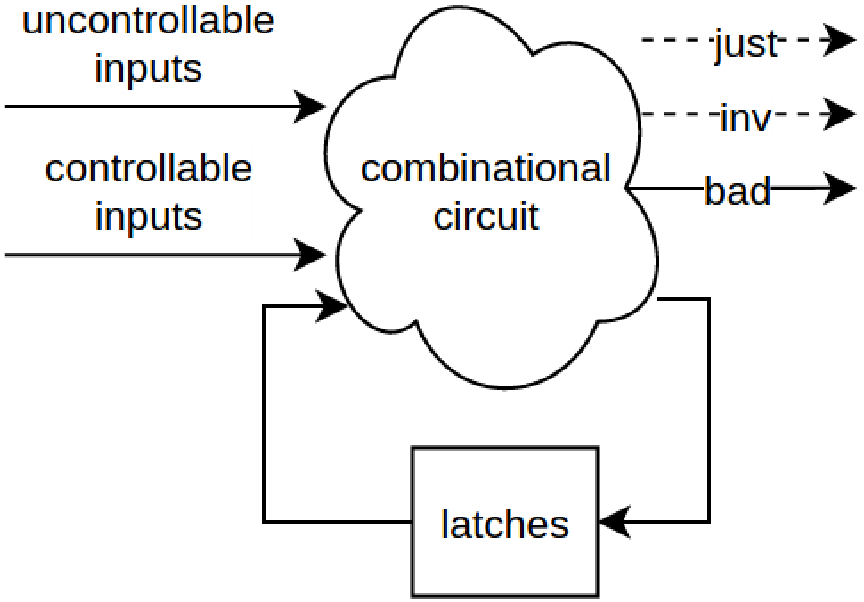
Only `main` can have specifications

LTL, LDL, RE, patterns? relations?

only safety assumptions

TRANSLATION INTO SYNTCOMP

SYNTCOMP format

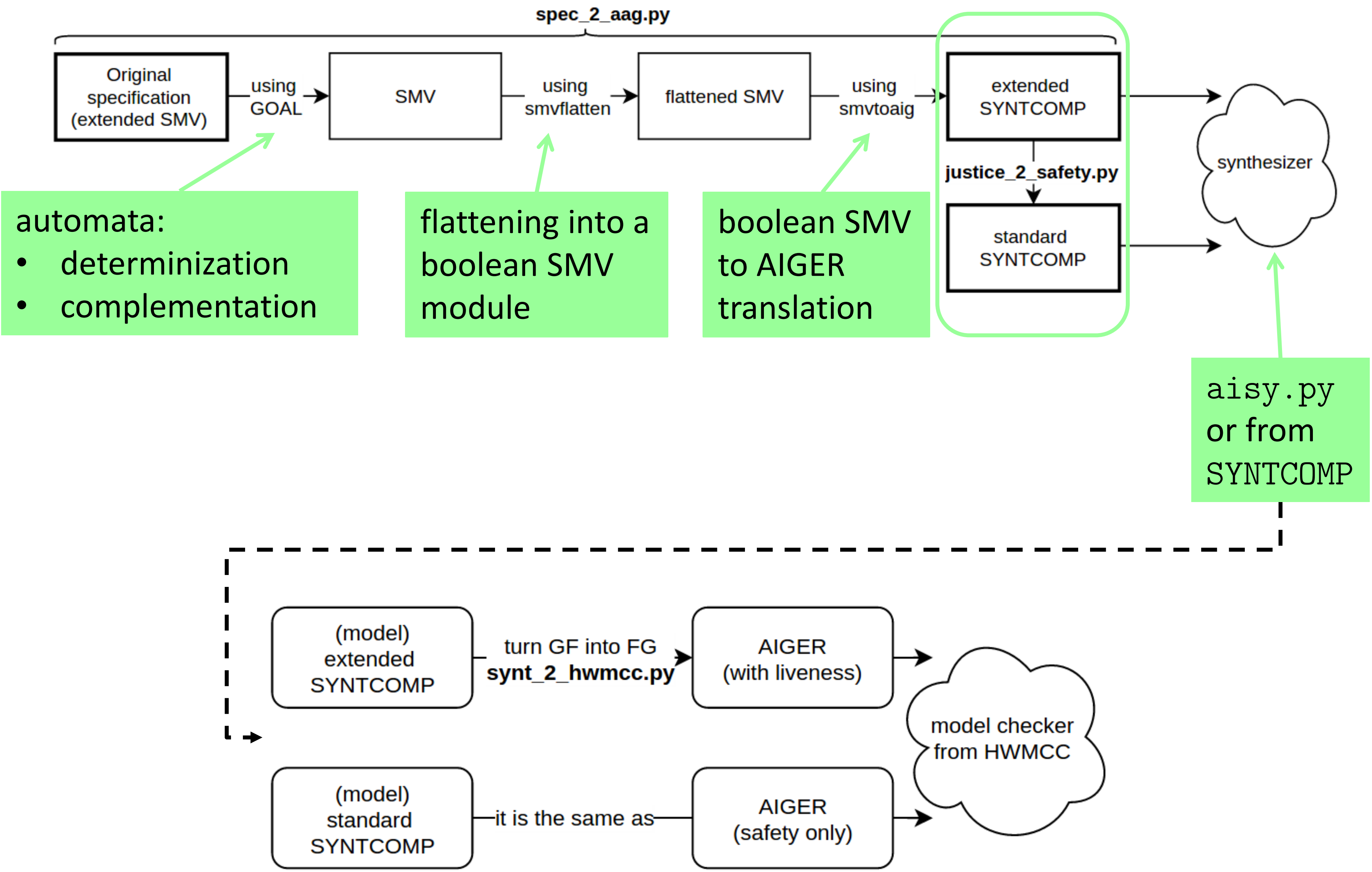


Standard: $\mathbf{G}\neg bad$

Extended with liveness:

$$(\neg bad\ \mathbf{W}\ \neg inv) \wedge (\mathbf{G}\ inv \rightarrow \mathbf{GF}\ just)$$

Working flow



automata:
- determinization
- complementation

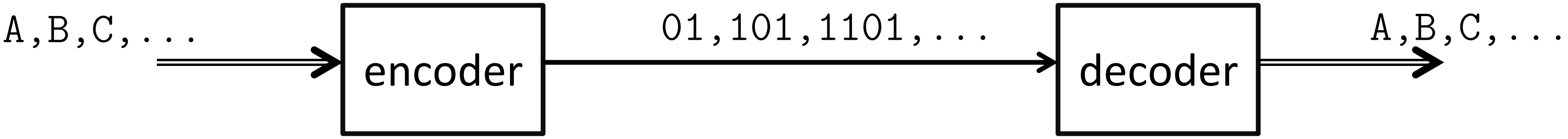
flattening into a boolean SMV module

boolean SMV to AIGER translation

`aisy.py` or from `SYNTCOMP`

SYNTHESIZING HUFFMAN ENCODER

Huffman encoding

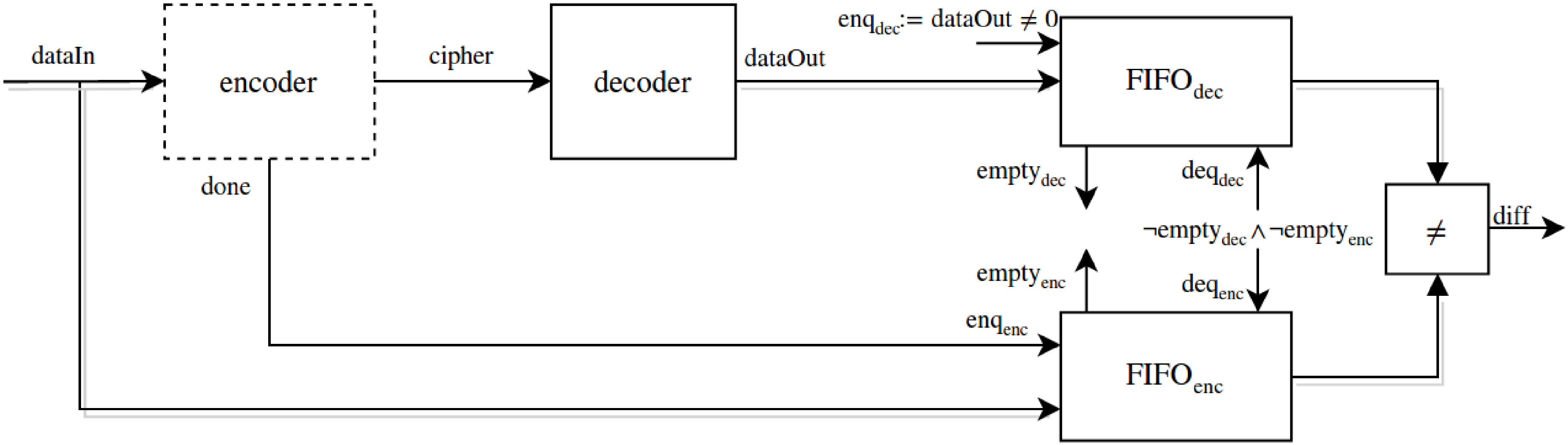


“more often appearing letters have shorter ciphers”

Letters frequency table

```
                      +-------------( )---------------+
                      |                               |
             +-------( )------+               +------( )-----+
             |                |               |              |
             |                |               |              |
       +----( )----+         ( )          +--( )--+         ( )
       |           |         / \          |       |         / \
       |           |        |   |         |       |        |   |
   +--( )--+      ( )      [E] ( )       ( )     ( )      [ ] ( )
   |       |      / \          / \       / \     / \          / \
   |       |     |   |        |   |     |   |   |   |        |   |
  ( )     ( )   [S] ( )      ( ) [A]   [I] [O] [R] [N]      ( ) [T]
  / \     / \       / \      / \                            / \
 |   |   |   |     |   |    |   |                          |   |
[U] [P] [F] [C]   ( ) [L]  [H] ( )                        [D] ( )
                  / \          / \                            / \
                 |   |        |   |                          |   |
           +----( ) [W]      [G] [Y]                        ( ) [M]
           |      \                                         / \
           |       |                                       |   |
          ( )     ( )                                     [B] [V]
          / \     / \
         |   |   |   |
        [Q] ( ) [K] [X]
            / \
           |   |
          [Z] [J]
```

Synthesizing a Huffman encoder



Specification

A1. “input $dataIn$ is within range 1..27”
A2. “$dataIn$ does not change until incl. the moment when $done$ is high”

G1. $\mathbf{G}(done \rightarrow \mathbf{X}\, enq_{dec})$
G2. $\mathbf{G}\, \neg diff$
G3. $\mathbf{GF}\, done$

Info about the synthesis

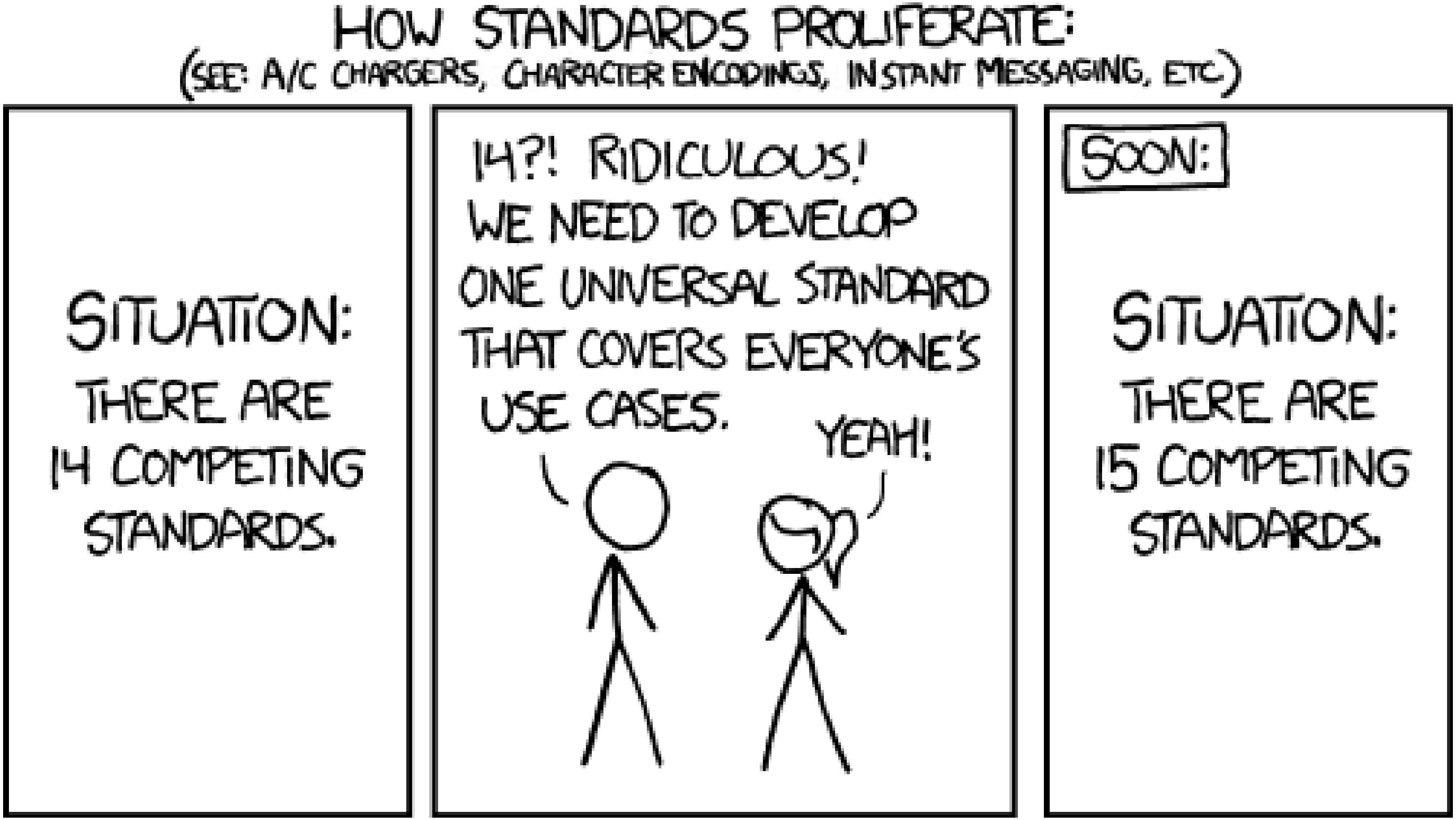
- The specification:
  - # latches = 45
  - # AND gates = 3k
- The model has:
  - # AND gates = 130k (120k)
- Timings:
  - 2min (4min)
- The model is as expected

Conclusion & discussion

- Adapted the SMV format to synthesis tasks
- Provided scripts to translate into the SYNTCOMP

- Is SMV good enough or Verilog should be used?
- Should we support LTL/RE formats?
- Should we support GR1 or full LTL semantics?
- Should we support partial information?
- Simpler ways to translate?

thank you


HOW STANDARDS PROLIFERATE:
(SEE: A/C CHARGERS, CHARACTER ENCODINGS, INSTANT MESSAGING, ETC)
SITUATION:
THERE ARE 14 COMPETING STANDARDS.
14?! RIDICULOUS! WE NEED TO DEVELOP ONE UNIVERSAL STANDARD THAT COVERS EVERYONE'S USE CASES.
YEAH!
SOON:
SITUATION:
THERE ARE 15 COMPETING STANDARDS.