Property Mining using Dynamic Dependency Graphs

Jan Malburg, Tino Flenker, and Görschwin Fey

Motivation

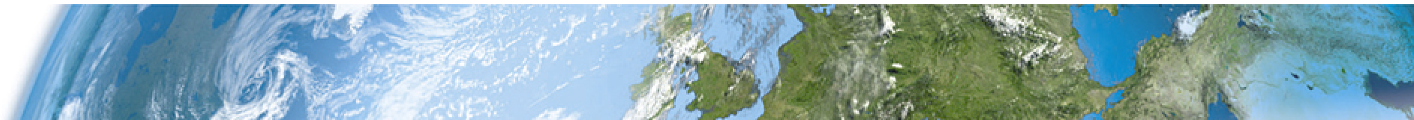
Problems in modern chip design:

- Complex designs (MSoCs)
- Large teams (>100 people)
- Tight Time-To-Market constraints
- Third party IP
- Safety critical systems
- Security

Automatic property generation

Using automatically generated properties for

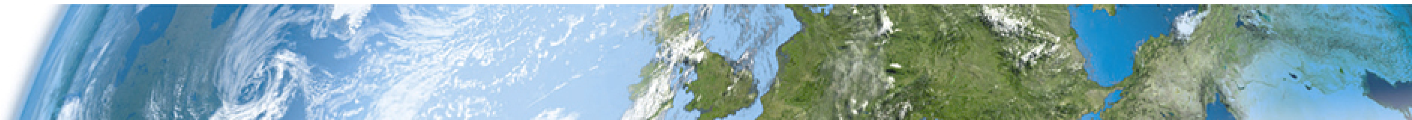
- Design understanding
- Automatic assertions
- Regression tests
- Pre-processing for other verification tasks
- Comparison with specification



State of the art

Dynamic property mining

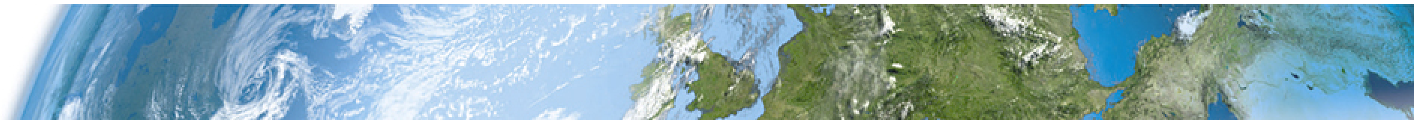
- Examples: GoldMine[DATE'10] , Daikon[TSE'01]
    - Use a set of templates to find relations between values observed during simulation
    - (If wanted) model checking generated properties
- Problems
    - Limited to templates or simple relations
    - Requires much simulation data
    - May link totally unrelated signals if simulation data is bad



Goal

Dynamic property mining approach that

- Requires only few simulation data
- Needs no templates
- Yields good quality properties
- Yields general properties



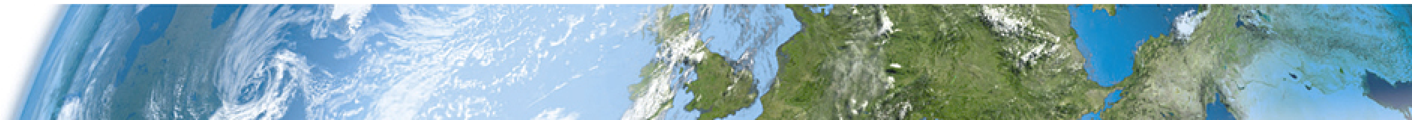
Steps

Use Cases → DDG Generation → Initial properties → Repetition folding → Property combi-nation → Property splitting → Property abstraction → Valid?

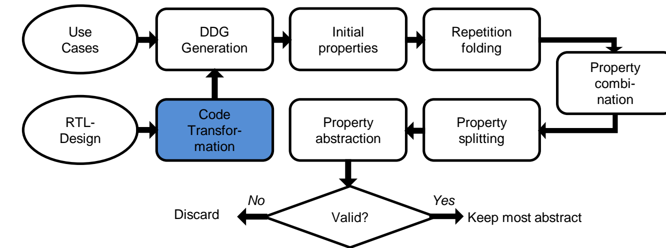
RTL-Design → Code Transfor-mation → DDG Generation

Valid? *No* → Discard

Valid? *Yes* → Keep most abstract



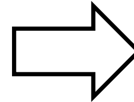
Code transformation for relevant dependency



- Relevant dependency: A variable is dependent on a control-statement, if another evaluation of the control-statement would result in another variable value
- Idea: Add a self-assignment, whenever a variable could be assigned

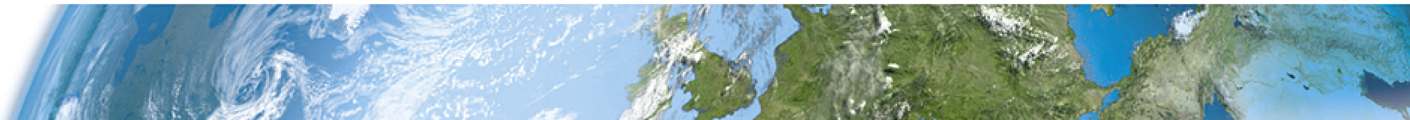
Design

```
module
ConditionalFlipFlop(
  input  wire clk,
  input  wire enable,
  input  wire dataIn,
  output reg  dataOut)

  always @(posedge clk)
    if(enable)
      dataOut <= dataIn;
endmodule
```

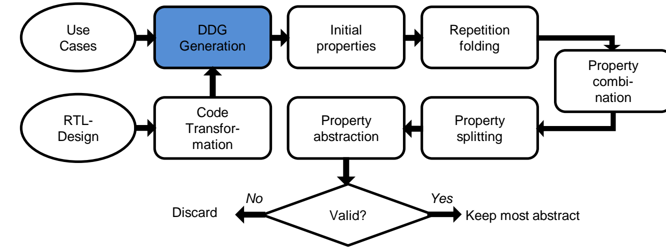


Transformed design

```
module
ConditionalFlipFlop(
  input  wire clk,
  input  wire enable,
  input  wire dataIn,
  output reg  dataOut)

  always @(posedge clk)
    if(enable)
      dataOut <= dataIn;
    else
      dataOut = dataOut;
endmodule
```



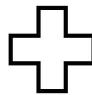
Creating dynamic dependency graphs



- Instrumentation: Signals replaced by SystemVerilog-Objects, statements and expression with function calls
- Dynamic dependency graph is stored during execution

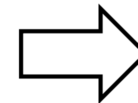
Transformed design

```
module
ConditionalFlipFlop(
  input  wire clk,
  input  wire enable,
  input  wire dataIn,
  output reg  dataOut)

  always @(posedge clk)
    if(enable)
      dataOut <= dataIn;
    else
      dataOut = dataOut;
endmodule
```

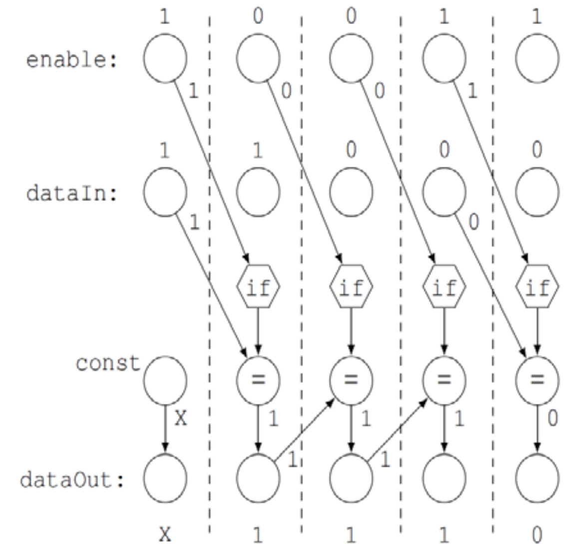


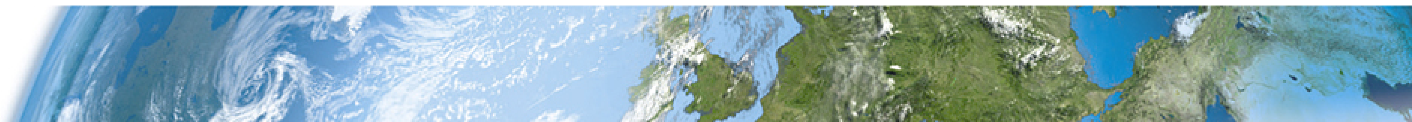
Use Case

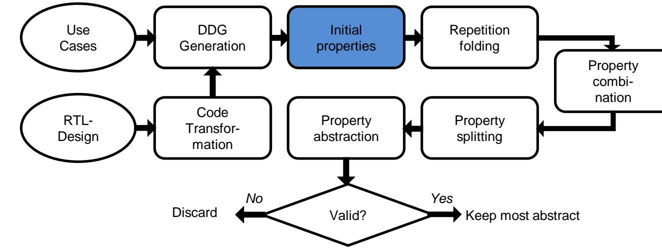
Clock cycle	0	1	2	3	4
enable	1	0	0	1	1
dataIn	1	1	0	0	0



DDG



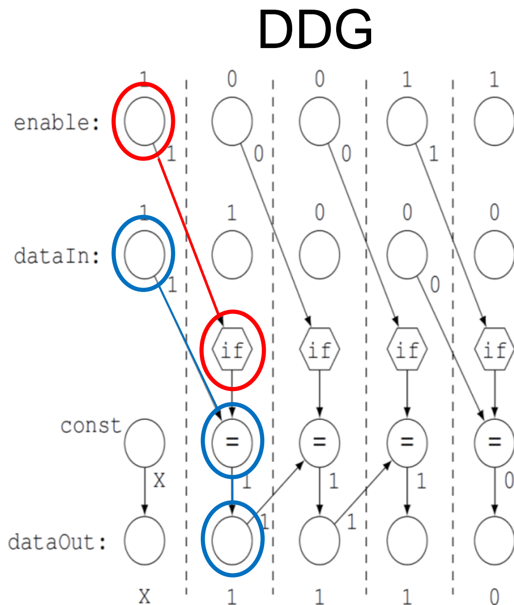




Computing path-constraints and symbolic expressions

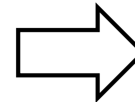
- Property format:
  <*path-constraint*> |-> <var> == <*symbolic expression*>
- Each control-statement on the path yields a path-condition
- Conjunction of path-conditions → path-constraint
- Data-path yields symbolic expression

DDG



Path-constraint:
enable==1

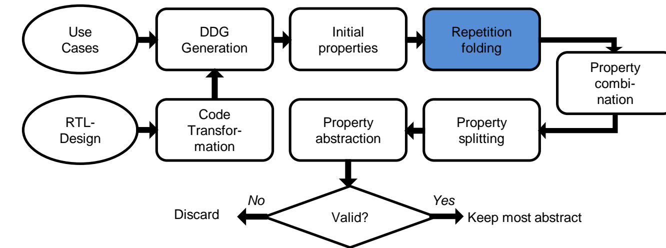
Symbolic-expression:
$past(dataIn,1)



Initial property

```
enable==1 |->
##1 dataOut ==
    $past(dataIn,1)
```

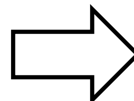
Folding repetitions within path-constraints



- Use the `[*c]`- operator to combine several identical path-conditions within a path-constraint
- Path-condition must be follow each other immediately in time
- Purely syntactic change

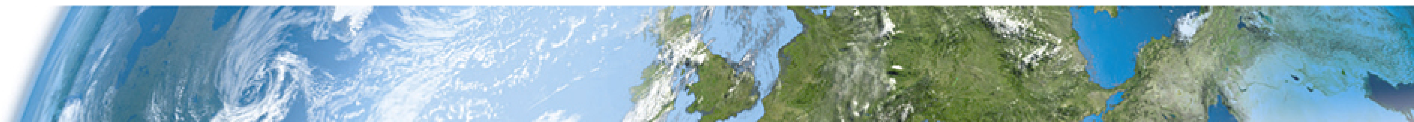
Initial property

```
enable == 1 ##1
enable == 0 ##1
enable == 0 ##1
enable == 0 |->
##1 dataOut ==
  $past(dataIn,4)
```

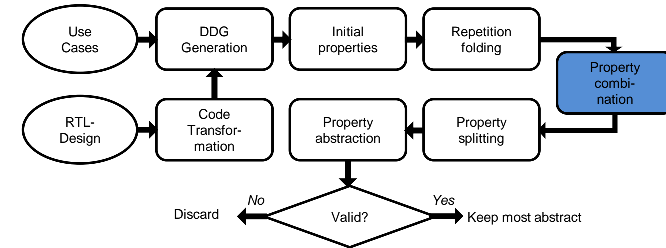


Folded property

```
enable == 1 ## 1
enable == 0 [*3] |->
##1 dataOut ==
  $past(dataIn,4)
```



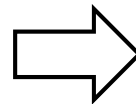
Combining similar properties



- Combine properties which only differ with respect to the constant in the `[*c]`- operator
- Use the `[*a:b]`- operator
- Not all repetitions in the range need to be hit
- Compute a support value: $\frac{|observed\ repetitions|}{elements\ in\ range}$, as quality measurement

Folded properties

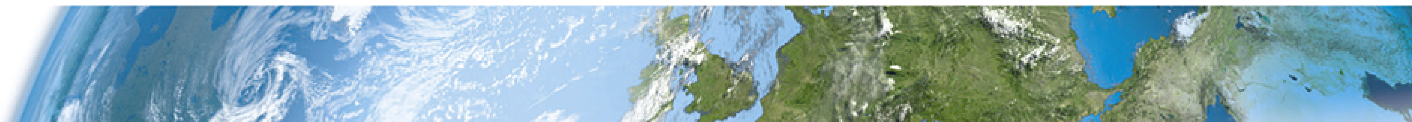
```
req[* 8] |-> ##1 ack
req[* 9] |-> ##1 ack
req[*11] |-> ##1 ack
req[*12] |-> ##1 ack
```



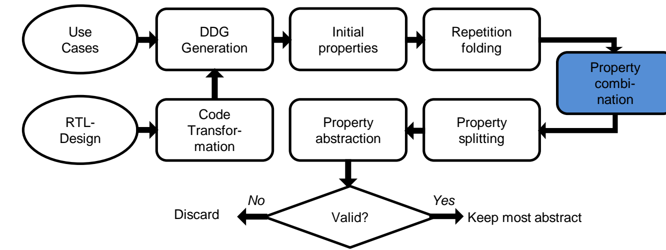
Combined property

```
req[*8:12] |-> ##1 ack
```

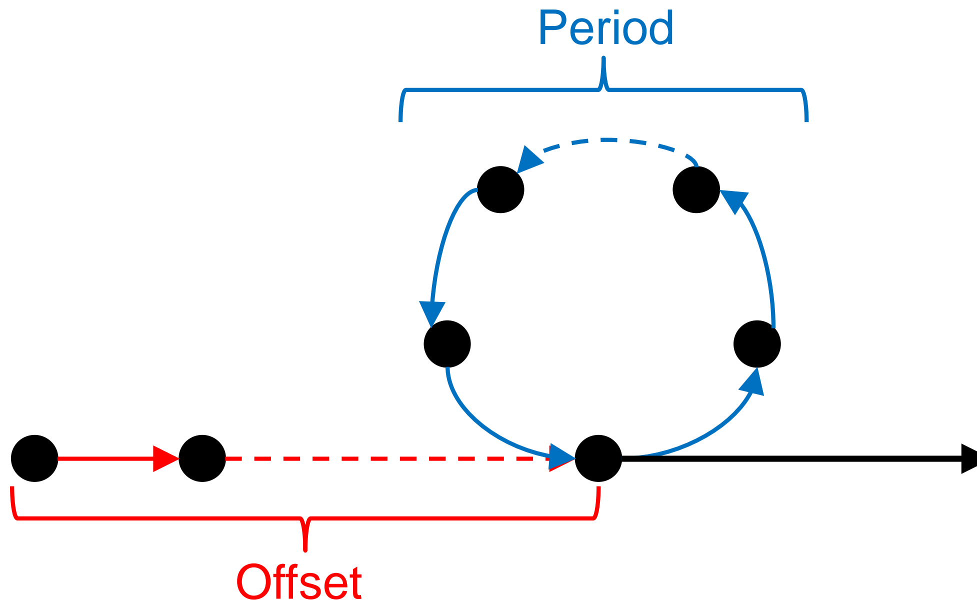
Support value: 0.8

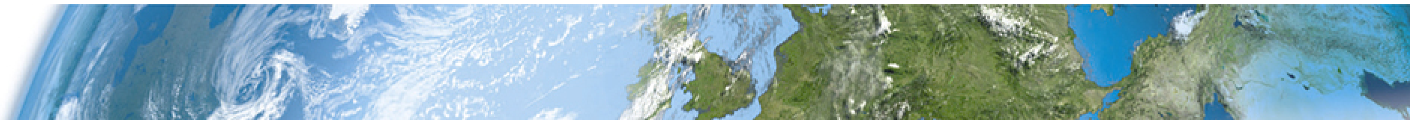


Combining similar properties (cont.)

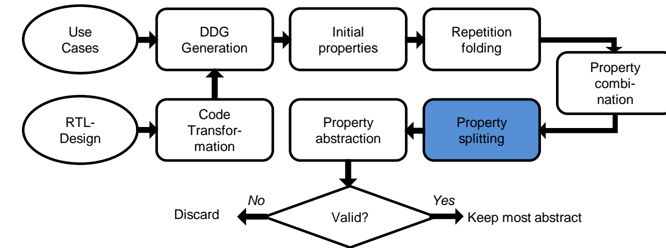


- To describe cases with periodic repetitions
  - Period: `c`
  - Offset: `d`
- Extended repetition format: `<path-condition>[*d] ##1 ( <path-condition>[*c] )[*a:b]`





Splitting properties

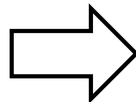


- Split properties using larger period values `c` to create properties with better support
- New properties have different offsets to cover all previous values
- Discard properties with support value = 0
- Keep property set with highest average support

Combined property

Split properties

```
req[*8:12]|->##1 ack
```

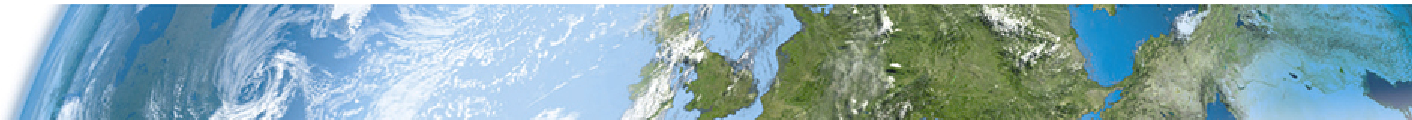


Support = 0 → Discard

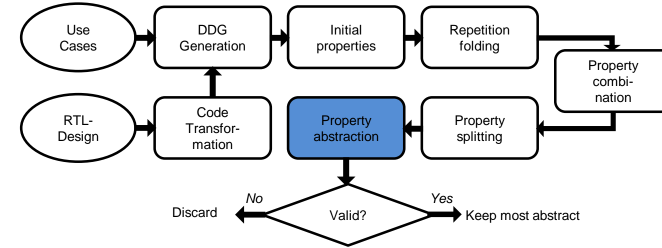
```
req[*1] ##1 (req[*3])[*3:3]|->##1 ack
```

```
req[*2] ##1 (req[*3])[*2:3]|->##1 ack
req[*3] ##1 (req[*3])[*2:3]|->##1 ack
```

Support = 1 → Keep



Abstracting properties



The range-limits of the `[*a:b]`- operator are moved to create more abstract properties

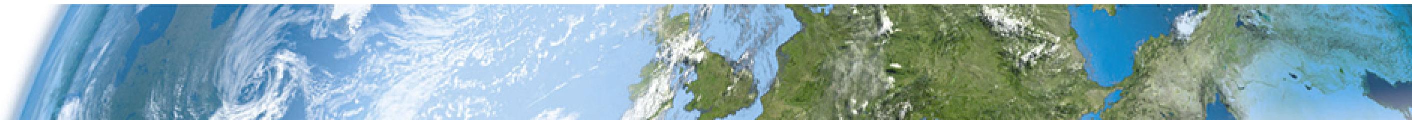
- `a` → `0`
- `a` → `1`
- `b` → `$` (∞)

Split property

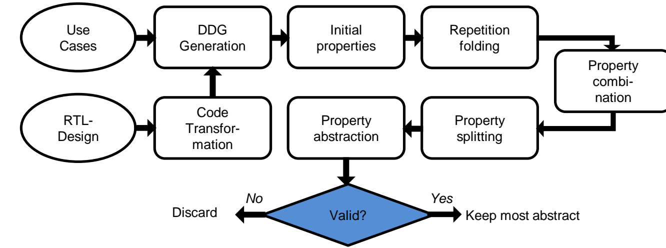
```
req[*2]##1(req[*3])[*2:3]|->##1 ack
```

Abstracted properties

```
req[*2]##1(req[*3])[*2:3]|->##1 ack
req[*2]##1(req[*3])[*1:3]|->##1 ack
req[*2]##1(req[*3])[*0:3]|->##1 ack
req[*2]##1(req[*3])[*2:$]|->##1 ack
req[*2]##1(req[*3])[*1:$]|->##1 ack
req[*2]##1(req[*3])[*0:$]|->##1 ack
```



Model checking



- The set of generated properties is checked
- For a set of abstracted properties only the most abstracted and correct property is kept

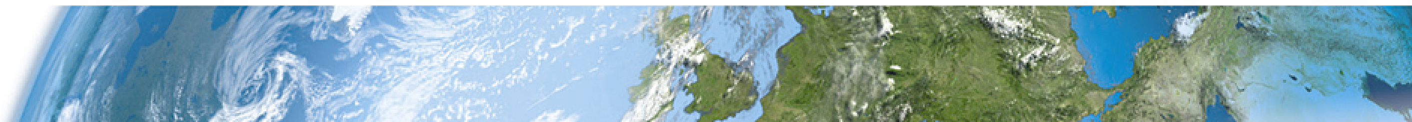
Abstracted properties

Final property

```
req[*2]##1(req[*3])[*2:3]|->##1 ack
req[*2]##1(req[*3])[*1:3]|->##1 ack
req[*2]##1(req[*3])[*0:3]|->##1 ack
req[*2]##1(req[*3])[*2:$]|->##1 ack
req[*2]##1(req[*3])[*1:$]|->##1 ack
req[*2]##1(req[*3])[*0:$]|->##1 ack
```

req[*2]##1(req[*3])[*1:$]|->##1 ack (bold above)

⇨ req[*2]##1(req[*3])[*1:$]|->##1 ack



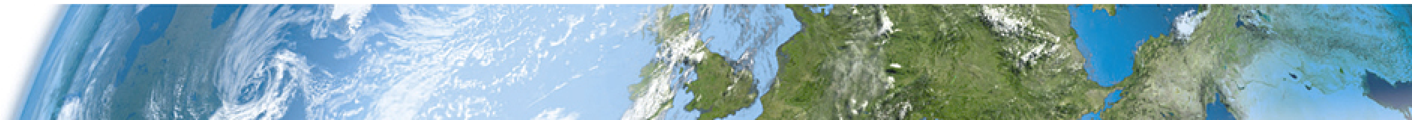
Small example

- Conditional count (8-Bit)
- Inputs:
    - enable
    - reset
- Output:
    - out
- Counts up when enable is true
- Generated properties:

```
property out_0;
  @ (posedge clk)
    (
     (reset)
    )
    |->##1 out ==8'b0;
endproperty
```

```
property out_1;
  @ (posedge clk)
    (
     (enable && !reset)
    )
    |->##1 out
==($past(out,1)+8'b1);
endproperty
```

```
property out_3;
  @ (posedge clk)
    (
     (!enable && !reset)
    )
    |->##1 out
==$past(out,1);
endproperty
```

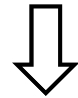


Experiments: Y86-CPU

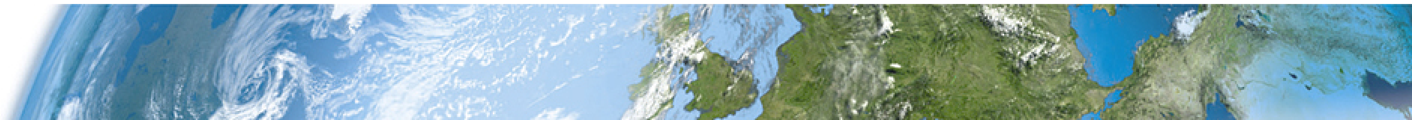
- x86-subset ISA
- Use case: 417 clock cycles
- Runtime:
  - 12.5s generation
  - 265m model checking
- Properties:
  - 171 before abstraction
  - 251 in total
  - 88 correct
  - 163 incorrect
  - 58 in final set

```
property bus_RE_i_5;
  @ (posedge clk)
    (
     (rst)
      ##1 ((!rst)[*5] )[*1:$]
    )
    |->##1 bus_RE =='b1;
endproperty

property bus_RE_i_2;
  @ (posedge clk)
    (
     (rst)
      ##1 (!rst)[*3] ##0((##1((!rst)[*5] )[*1:$]) or 1 )      //(!rst)[*3 + n*5]  ,n>=0
    )
    |->##1 bus_RE ==((current_opcode=='b10001011)&(({$past(bus_in,3)}[15:14])=='b1));
endproperty
```

Better human readable version as comment

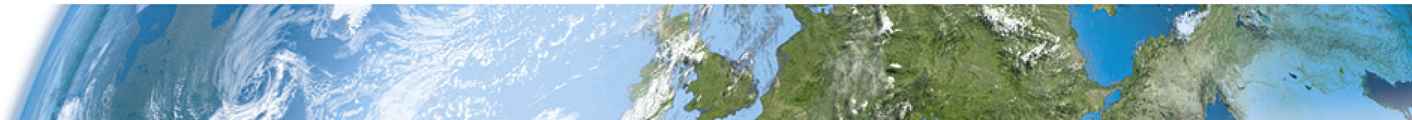


Read-OpCode



Comparison with GoldMine

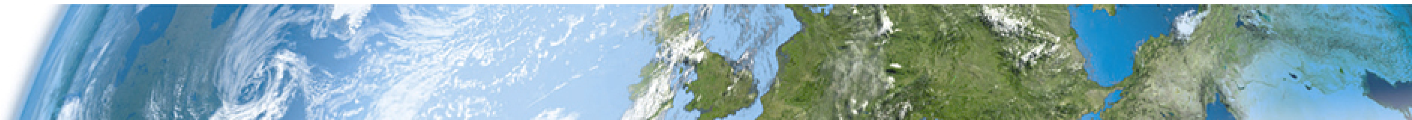
	GoldMine	Our Approach
Required sim. Clock cycles	Many	Few
Interesting internal signals	Found by heuristic	Manual
Asynchronous behaviour	Yes	No
Multi-bit properties	No	Yes
Expressions in properties	==(1|0)	All, except "?:"
Equivalence between variables	No	Yes
Temporal unbounded properties	No	Yes



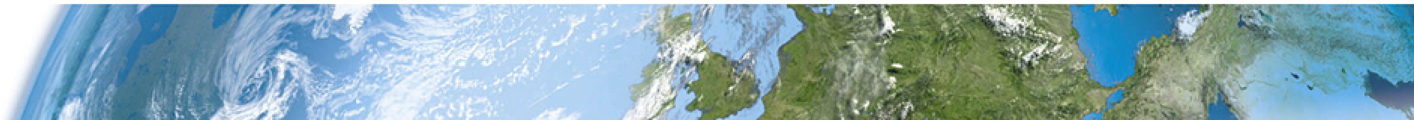
Conclusion

Automatic property generation

- Basic idea
  - Extract initial properties from dynamic dependency graphs
  - Abstract initial properties to general properties
  - Model check
- Small/few use cases suffice for good properties
- Expressions extracted from the dynamic dependency graphs
  - ➢ no expression template required



Thank you for your attention



References

[DATE'10] Shobha Vasudevan, David Sheridan, Sanjay Patel, David Tcheng, Bill Tuohy, and Daniel Johnson, "*GoldMine: Automatic assertion generation using data mining and static analysis*", in Design, Automation and Test in Europe, 2010

[TSE'01] M. Ernst, J. Cockrell,W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," IEEE Transactions on Software Engineering, vol. 27, no. 2, pp. 99–123, 2001.