Note: content of these slides are from "Getting started with SMV" by K. L. McMillan, 1999. Refer to this document for more details.

SMV TUTORIAL – Part II

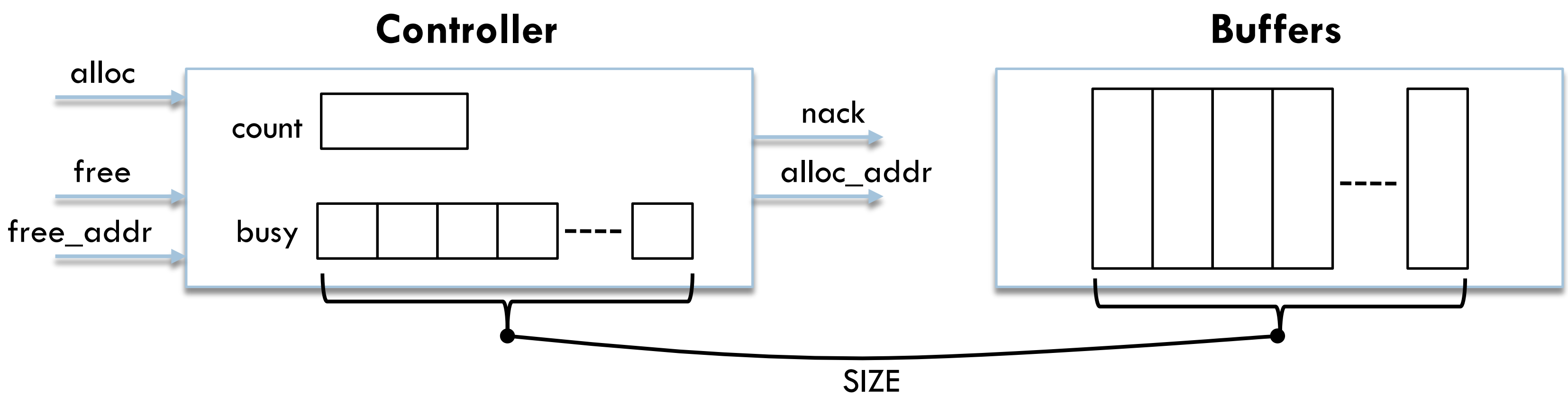
Eriko Nurvitadhi

Agenda

- Part I – SMV Basics
- Part II – Compositional Verification (*this talk*)
  - Compositional verification overview
    - Example 1: using layer
    - Example 2: decomposition
  - Compositional techniques
    - Array transmitter example
    - Creating abstract model and a simple implementation
    - Multi-stage implementation
    - Refinement maps as types
    - Decomposition benefit
    - Decomposing large data structures
    - Exploiting symmetry
    - Case analysis
    - Data type reduction
    - Induction

Last time: BDD Limit

- Example: controlling the allocation and freeing of buffers



- Verifying `safe[0]`
  - Buffer size of 32 → under 1 minute, 10^9 states reached
  - Buffer size of 64 → 10+ minutes, 10^{19} states reached
  - … eventually, state explosion!

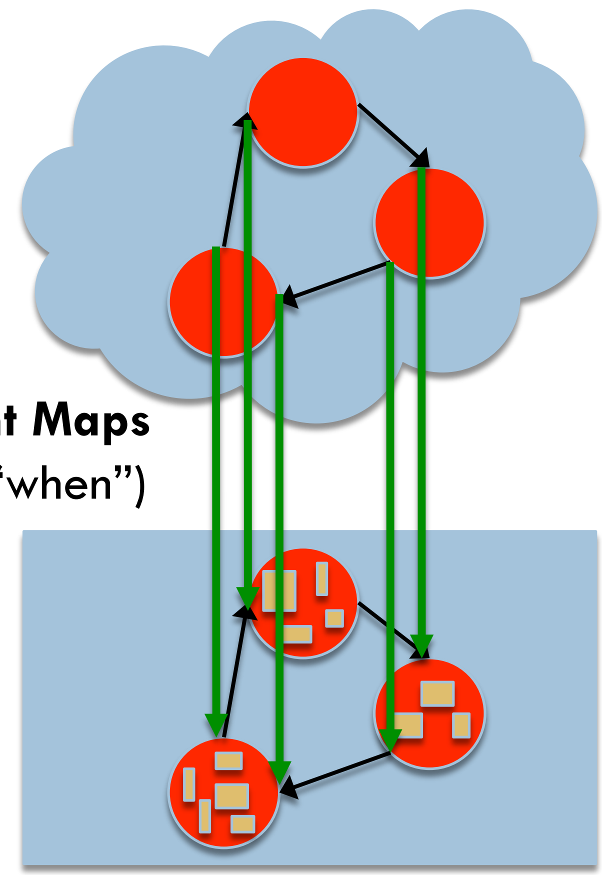
BDD helps, but doesn't *eliminate* state explosion. How can we scale further?

Compositional Verification

Abstract Model
("what")
simple

Refinement Maps
("where", "when")

Implementation
("how")
sophisticated

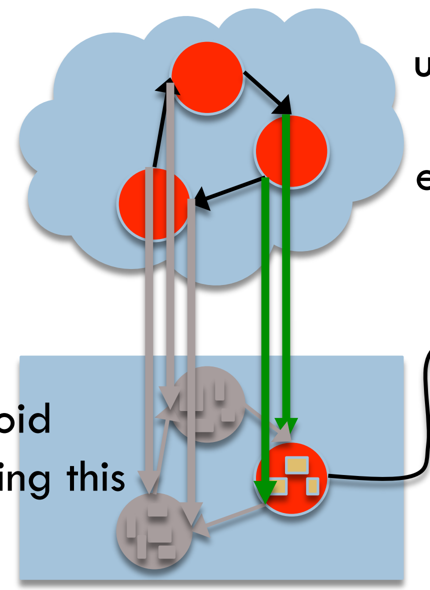


Compositional Verification
Verify each part of implementation using environment of abstract model
→ *avoids model checking the entire implementation in one shot*

use abstract model's environment

verify this

avoid exploring this

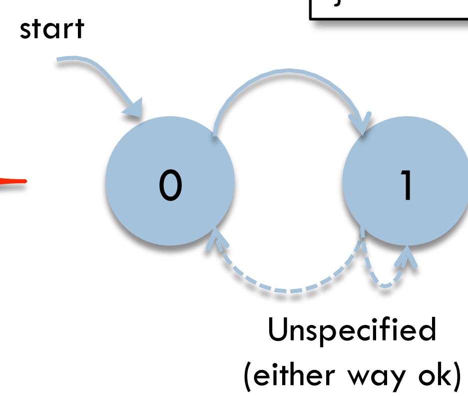


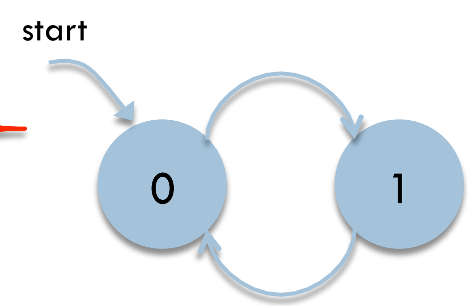
Example 1: Using Layer

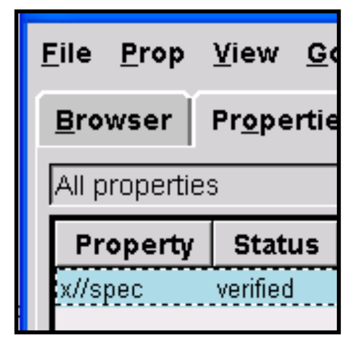
Layer: abstract signal definitions

```
layer <layer_name> : {
    assignment1;
    assignment2;
    ...
    assignmentn;
}
```

```
module main(){
   x : boolean;

   /* the specification */

   layer spec: {
     init(x)            := 0;
     if(x=0) next(x) := 1;
     else    next(x) := {0,1};
   }

   /* the implementation */

   init(x) := 0;
   next(x) := ~x;
 }
```







Example 2: Decomposition

Property	Status
x//spec	verified
y//spec	verified

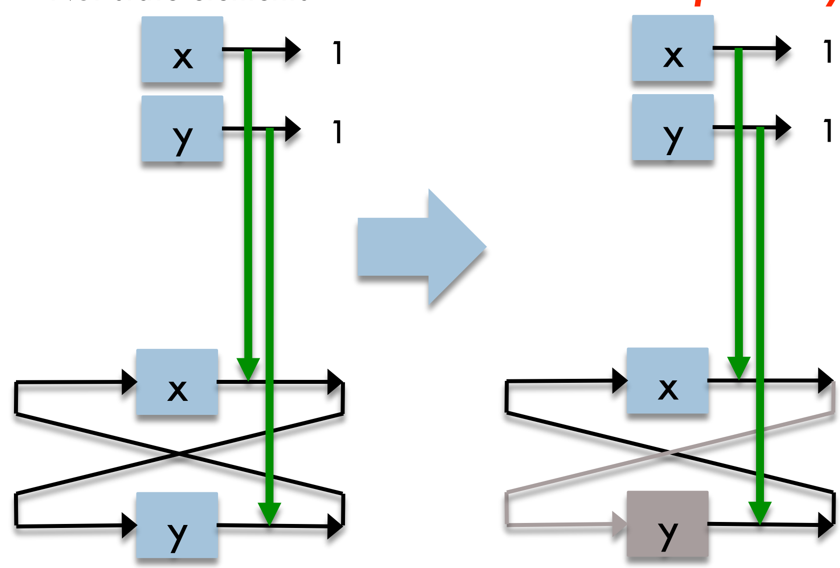
Entire cone | State vars: 1, Comb. vars: 0

Signal	Layer	Vars	Type
x	.	1	state
y	spec		

```
module main(){
   x,y : boolean;

   /* the specification */

   layer spec: {
     x := 1;
     y := 1;
   }

   /* the implementation */

   init(x) := 1;
   next(x) := y;
   init(y) := 1;
   next(y) := x;
}
```

Decomposition: x, y verified *separately!*

Not state elements



state elements init to 1

To verify x, no need to consider y at implementation-level → model check only 1 state element instead of 2!

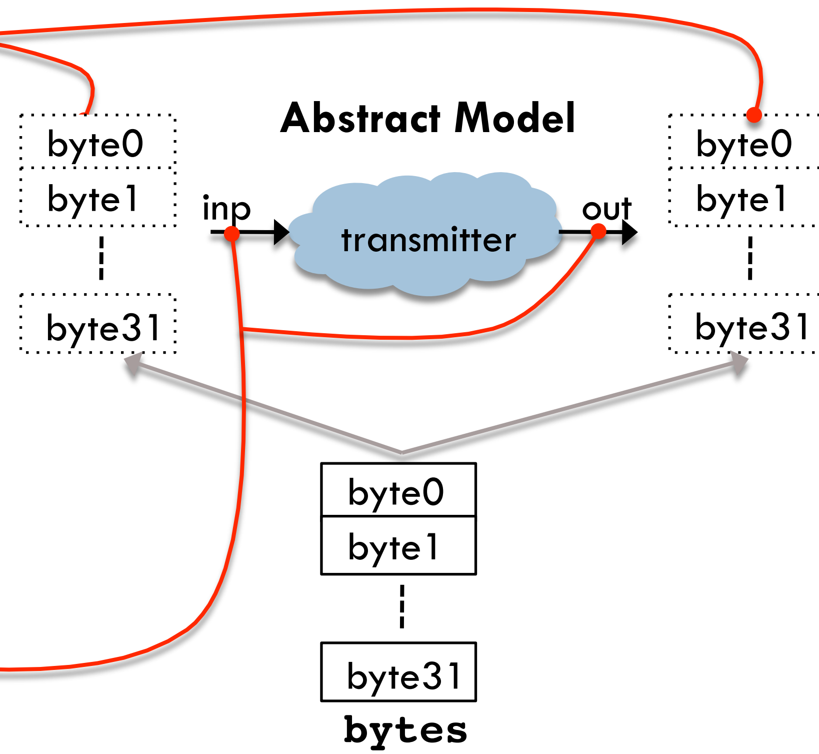
Agenda

- Part I – SMV Basics
- Part II – Compositional Verification (*this talk*)
  - Compositional verification overview
    - Example 1: using layer
    - Example 2: decomposition
  - **Compositional techniques**
    - Array transmitter example
    - Creating abstract model and a simple implementation
    - Multi-stage implementation
    - Refinement maps as types
    - Decomposition benefit
    - Decomposing large data structures
    - Exploiting symmetry
    - Case analysis
    - Data type reduction
    - Induction

Array Transmitter Example


```
typedef BIT 0..7;
typedef INDEX 0..31;
typedef BYTE array BIT of boolean;

module main(){

  /* the abstract model */

  bytes : array INDEX of BYTE;
  next(bytes) := bytes;

  /* the input and output signals */

  inp, out : struct{
    valid : boolean;
    idx : INDEX;
    data : BYTE;
  }
```

Circuit to transmit an array of 32 bytes from input to output (no operations performed on array)



Abstract and Implementation Models

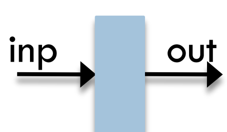
- Abstract model
  - contents of input array is same as output (i.e., `bytes`)
  - transmit when `valid`, based on `idx`; "don't care" otherwise

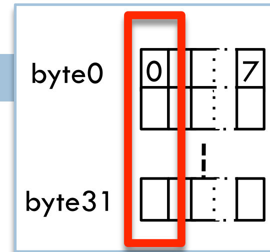
```
/* the refinement maps */
 layer spec: {
   if(inp.valid) inp.data := bytes[inp.idx];
   if(out.valid) out.data := bytes[out.idx];
 }
```

- (Very simple) Implementation
  - input to output, delayed 1 time unit

```
 /* the implementation */
 init(out.valid) := 0;
 next(out) := inp;
}
```

Implementation





All properties

Property	Status
out.data[0]//spec	verified
out.data[1]//spec	verified
out.data[2]//spec	verified
out.data[3]//spec	verified
out.data[4]//spec	verified
out.data[5]//spec	verified
out.data[6]//spec	verified
out.data[7]//spec	verified

39 state vars
- 1 per 32 elements of bytes[0][0] to bytes[31][0]
- 5 for out.idx, 1 for out.data[0], 1 for out.valid

6 Comb vars for 'free' inp (5 for idx, 1 for valid)

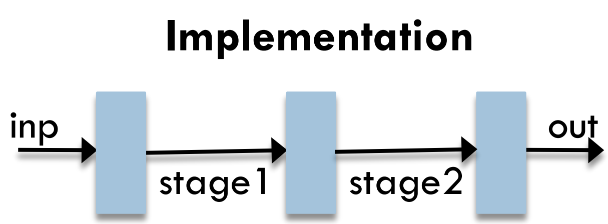
Entire cone — State vars: 39, Comb. vars: 6

Signal	Layer	Vars	Type
bytes[8][0]	.	1	state
bytes[9][0]	.	1	state
inp.data[0]	spec		
inp.idx	free	5	comb
inp.valid	free	1	comb
out.data[0]	.	1	state
out.idx	.	5	state
out.valid	.	1	state

Multi-Stage Implementation

- Suppose we have implementation with multiple stages
  - want to verify correct transmission
  - don't want to model check entire multi-stage; verify ***individual*** stages instead

```
stage1, stage2 : struct {
  valid : boolean;
  idx : INDEX;
  data : BYTE;
}

init(stage1.valid) := 0;
next(stage1)       := inp;
init(stage2.valid) := 0;
next(stage2)       := stage1;
init(out.valid)    := 0;
next(out)          := stage2;
```



Multi-Stage Implementation

properties = 8 x 3 = **24** (i.e., for bit 0 to 7, and for out.data[], stage1.data[], and stage2.data[])

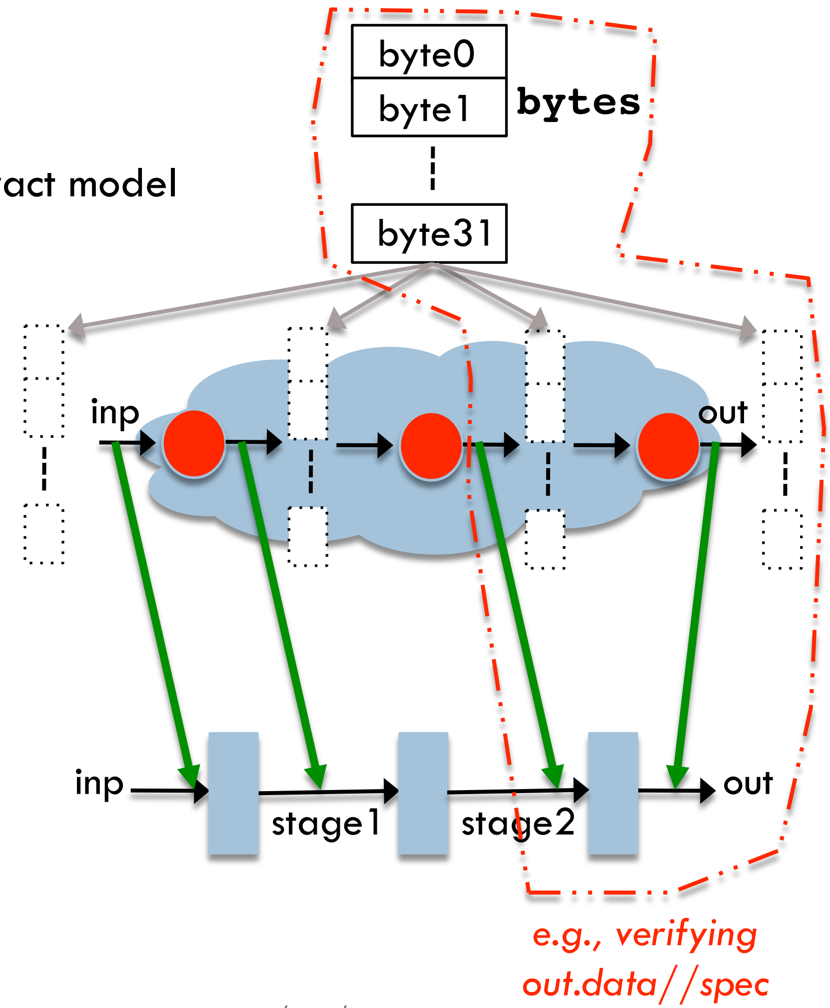
39 state vars per property, since each stage verified **independently**

- refinement map
  - verifies a stage output with input from abstract model

```
layer spec: {
   if(stage1.valid)
      stage1.data := bytes[stage1.idx];
   if(stage2.valid)
      stage2.data := bytes[stage2.idx];
 }
```

- free stage2's `valid` and `idx`
  - makes last stage verification independent
  - can also do the same for stage1

```
using
  stage2.valid//free, stage2.idx//free
prove
  out.data//spec;
```



Refinement Maps as Types

- Simplify description using a parameterized type, `module`
  - refinement maps can be specified inside of a module

```
module byte_intf(bytes){

    bytes : array INDEX of BYTE;

    valid : boolean;
    idx : INDEX;
    data : BYTE;

    layer spec:
      if(valid) data := bytes[idx];
}
```

Refinement Maps as Types


```
module main(){
  /* the abstract model */
  bytes : array INDEX of BYTE;
  next(bytes) := bytes;

  /* the input and output signals */
  inp, out : byte_intf(bytes);

  /* the implementation */
  stage1, stage2 : byte_intf(bytes);

  init(stage1.valid) := 0;
  next(stage1) := inp;
  init(stage2.valid) := 0;
  next(stage2) := stage1;
  init(out.valid) := 0;
  next(out) := stage2;

  /* abstraction choices */
  using stage2.valid//free, stage2.idx//free prove out.data//spec;
  using stage1.valid//free, stage1.idx//free prove stage2.data//spec;
}
```

Refinement maps are part of **`byte_intf`** ***data type***

Note: still same properties as before (24 total, 39 state vars per property)

We only simplified the SMV description using `module`

Decomposition Benefit

- Suppose we have 32 stages
  - see "Getting started with SMV" for the SMV code

- If we don't verify each stage independently
  - 8 properties (data[0..7] at output)
  - 256 state variables per property
    - 224 states from 7 states (5 idx, 1 valid, 1 bit data[]) for each of the 32 stages
    - 32 states for each element in array (e.g., bytes[0][0],.., bytes[31][0]]

  → state explosion

- If we verify each stage independently (as in prev slides)
  - 256 properties (8 properties for data[0..7] x 32 stages)
  - 39 state variables (verify each stage independently)

  → many properties, but each one is small and verifiable

Decomposing Large Data Structures

- For a large structure, would like to verify individual elements
  - Otherwise, entire structure is too large to verify
  - E.g., array size – 32 bytes vs. 1M bytes

```
module byte_intf(bytes){
    bytes : array INDEX of BYTE;
    valid : boolean;
    idx : INDEX;
    data : BYTE;

    forall(i in INDEX)
      layer spec[i]:
        if(valid & idx = i)
           data := bytes[i];
}
```

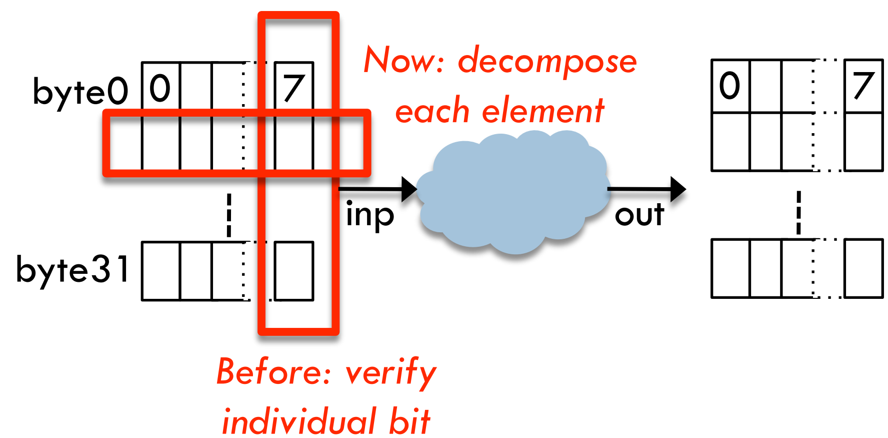
In main()

```
forall(i in INDEX){
   using
     stage2.valid//free, stage2.idx//free
   prove
     out.data//spec[i];

   using
     stage1.valid//free, stage1.idx//free
   prove
     stage2.data//spec[i];
}
```

Decomposing Large Data Structures

- Layer `spec[i]` is now an array
  - specifies the value of `data` only when `idx` is `i`, otherwise `data` is undefined
  - many abstract definitions for `inp.data` ***w/o*** implementation → not allowed
    - A fix: declare `inp` as an explicit input to the design
    - refinement maps at inputs are not verified, they are taken as assumptions

```
module main(bytes,inp,out){
  bytes : array INDEX of BYTE;
  input  inp : byte_intf(bytes);
  output out : byte_intf(bytes);
```

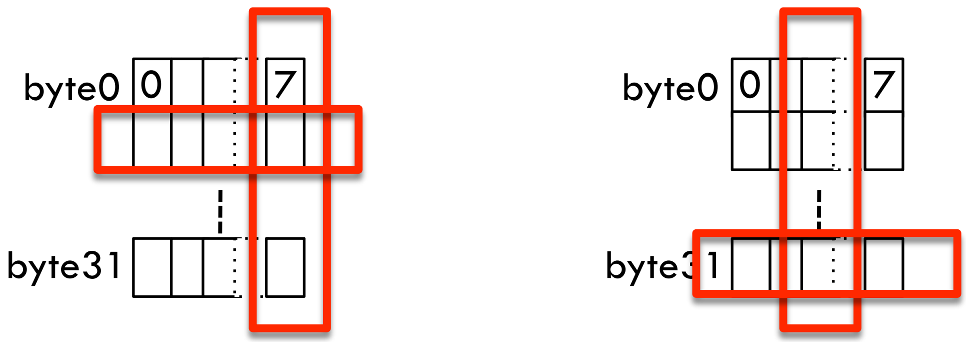


Every property **decomposed into 32 cases** (one for each byte in the array).

So, **32x more properties**, but each property **only has 8 state** vars (-31 vars since only consider 1 byte of the array per property)

Exploiting Symmetry

- Symmetry: equivalent states from verification perspective
  - can use `scalarset` to tell SMV where the symmetry is



- E.g., make INDEX symmetric
  - `typedef INDEX 0..31;` → `scalarset INDEX 0..31;`
  - Only need to verify 1 out of 32 array elements → 32x savings (# properties)
- E.g., make BIT symmetric
  - `typedef BIT 0..7;` → `scalarset BIT 0..7;`
  - Only need to verify 1 out of 7 bits, the rest is true by symmetry → 7x savings

Case Analysis

- Can decompose by cases considering each possible value of a variable
  - for the array transmitter example, we can consider each value of idx

```
scalarset BIT 0..7; scalarset INDEX 0..31; typedef BYTE array BIT of boolean;

module main(){
   bytes : array INDEX of BYTE;
   next(bytes) := bytes;

   inp, out : struct{
      valid : boolean; idx : INDEX; data : BYTE;
   }

   /* the refinement maps */
   layer spec: {
      if(inp.valid) inp.data := bytes[inp.idx];
      if(out.valid) out.data := bytes[out.idx];
   }

   /* the 1-stage implementation */
   init(out.valid) := 0;
   next(out) := inp;
}
```

Case analysis for idx:

```
forall (i in INDEX)
   subcase spec_case[i]
      of out.data//spec
      for out.idx = i;
```

Insead of

```
   out.data//spec
```

we now have

```
   out.data//spec_case[0]
   …
   out.data//spec_case[31]
```

If each `spec_case[]` is true, then `spec` must be true

Data type reduction

- Suppose we're verifying large or unknown size of array
  - SMV can employ an abstract type, with a small fixed number of values
  - have an abstract value to represent ***remaining*** values in original type

- For example: verifying bytes with idx of i from 0..31
  - for byte i, can reduce idx type to ***i*** and ***all other numbers not*** i (NaN)
  - note: NaN == NaN → undetermined value

=	**i**	**Nan**
i	1	0
Nan	0	{0,1}

Data Type Reduction: Example


```
scalarset BIT 0..7; scalarset INDEX 0..31; typedef BYTE array BIT of boolean;

module main(){
   bytes : array INDEX of BYTE;
   next(bytes) := bytes;

   inp, out : struct{
      valid : boolean; idx : INDEX; data : BYTE;
   }

   layer spec: {  /* the refinement maps */
      if(inp.valid) inp.data := bytes[inp.idx];
      if(out.valid) out.data := bytes[out.idx];
   }

   init(out.valid) := 0; /* the implementation */
   next(out) := inp;

   forall (i in INDEX) /* case splitting */
      subcase spec_case[i] of out.data//spec
         for out.idx = i;
}
```

As is, ***5 state vars*** for to encode the 32 values of INDEX (0..31)

Notice for case `i`, if `idx != i`, we don't care what the output is since `spec_case[i]` only specifies `out.data` when `out.idx=i`

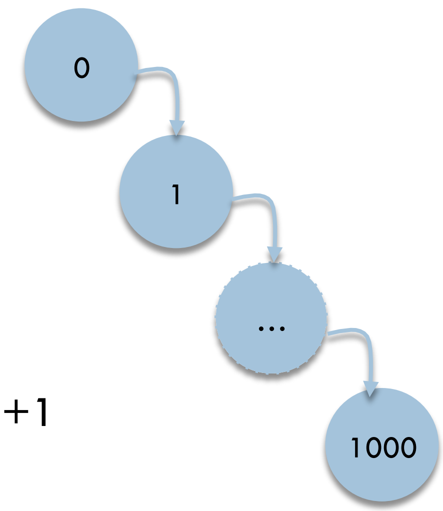
We can apply type reduction

```
forall (i in INDEX)
   using INDEX->{i} prove
      out.data//spec_case[i];
```

This reduces INDEX to abstract type of just value i and NaN

Now, ***only 1 state var*** to encode 2 values (i and NaN) of the abstract type of INDEX

Decomposing Long Sequence: Induction



- Suppose we want to verify property of a long sequence
  - e.g., counting up to a very large number
  - → deep state space, many iterations
- Proof by Induction
  - prove that property holds for 0 (base case)
  - prove that if it holds for some arbitrary value i, then it holds for i+1
- Use data type reduction for induction
  - suppose we're proving property p[i], where i is induction param ranging over TYPE
  - apply data type reduction that maps TYPE into X, i-1, i, Y
    - X: all values less than i-1; Y: all values larger than i
  - incrementing a value in this reduced type is defined as
    - X + 1 = {X, i-1}
    - (i-1) + 1 = i
    - i + 1 = Y
    - Y + 1 = Y

Induction Example


```
ordset TYPE 0..1000;

module main()
{
  x : TYPE;

  /* the counter */
  init(x) := 0;
  next(x) := x + 1;

  /* the property */
  forall(i in TYPE)
    p[i] : assert F (x = i);

  /* the proof */
  forall(i in TYPE)
    using p[i-1] prove p[i];
}
```

Want to show that for any value `i` from 0 to 1000, the counter eventually reaches `i`

SMV automatically generates a data type reduction using values `i` and `i-1`

And, it generates ***two*** representative cases to prove: `p[0]` and `p[2]`

Without induction, we would have had: `p[0], p[1], …, p[1000]`

What's Left

- More sophisticated examples
  - illustrate usages of compositional features discussed here
- Instruction processor examples
  - a simple in-order pipeline
  - an out-of-order pipeline
- Refer to SMV manual
  - "Getting started with SMV", K. L. McMillan, 1999