Combinatorial Test Automation Support for VDM++

Adriana Sucena Santos

Agenda

- Motivation
- Related work
- Introduction to Combinatorial Testing
- Specification
- Pros and cons
- Future work

Motivation

- Give more confidence to VDM++ models
- Help testing VDM++ models

Motivation

- Give more confidence to VDM++ models
- Help testing VDM++ models
- Avoid repetitive work

Motivation

- Give more confidance to VDM++ models
- Help testing VDM++ models
- Avoid repetitive work
- Enrich the Overture tool with Combinatorial Test Automation Support

Motivation

- Give more confidance to VDM++ models
- Help testing VDM++ models
- Avoid repetitive work
- Enrich the Overture tool with Combinatorial Test Automation Support
- Provide documentation about combinatorial testing applied to VDM++

Related work

- Tobias
  - VDM-SL
  - JML
  - Same theoretical principles

Introduction

- Idea: automatically generate the minimum number of test cases, testing the model exhaustively.
- How: regular expressions

Regular Expression symbols

- a +
- a *
- a ?
- a {n}
- a {n, m}
- a | b

VDM++ class groups

- types
- values
- operations
- ...
- **traces**

Stack example

```
class Stack
 instance variables
   stack : seq of nat := [];

 operations
public Push3 : nat ==> ()
Push3(e) ==
      stack := [e] ^ stack
   pre len stack < 3
   post stack = [e] ^ stack~;



end Stack
```

Stack example

```
class Stack
 instance variables
   stack : seq of nat := [];

 operations
public Push3 : nat ==> ()
Push3(e) ==
      stack := [e] ^ stack
  pre len stack < 3
   post stack = [e] ^ stack~;



end Stack
```

Stack example

```
class Stack
 instance variables
   stack : seq of nat := [];

 operations
public Push3 : nat ==> ()
Push3(e) ==
      stack := [e] ^ stack
   pre len stack < 3
   post stack = [e] ^ stack~;

public
   Pop : () ==> nat
   Pop() ==
     def res = hd stack in
       (stack := tl stack;
        return res)
   pre stack <> []
   post stack~ = [RESULT]^stack;

end Stack
```

Stack example

```
class Stack
  instance variables
    stack : seq of nat := [];

  operations
public Push3 : nat ==> ()
       …
public Pop : () ==> nat
       …
  traces
Push3(1){0,…,4}; Pop()


end Stack
```

Stack example

```
class Stack
  instance variables
    stack : seq of nat := [];

  operations
public Push3 : nat ==> ()
       ...
public Pop : () ==> nat
       ...
  traces
Push3(1){0,…,4}; Pop()

end Stack
```

1. Pop()
2. Push3(1); Pop()
3. Push3(1); Push3(1); Pop()
4. Push3(1); Push3(1); Push3(1); Pop()
5. Push3(1); Push3(1); Push3(1); Push3(1); Pop()

Stack example 2

```
class Stack
 instance variables
   stack : seq of nat := [];

 operations
public
   Push : nat ==> ()
   Push(e) ==
     stack := [e] ^ stack
   pre e < 10
   post stack = [e] ^ stack~;

public
   Pop : () ==> nat
   Pop() ==
     def res = hd stack in
       (stack := tl stack;
        return res)
   pre stack <> []
   post stack~ =
[RESULT]^stack;

end Stack
```

Stack example 2

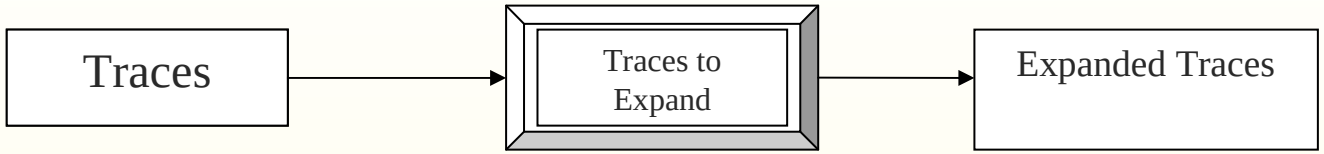
```
class Stack
  instance variables
    stack : seq of nat := [];

  operations
public Push : nat ==> ()
       …
public Pop : () ==> nat
       …
  traces
let x in set {1,5,10} in
Push(x); Pop()

end Stack
```

Stack example 2

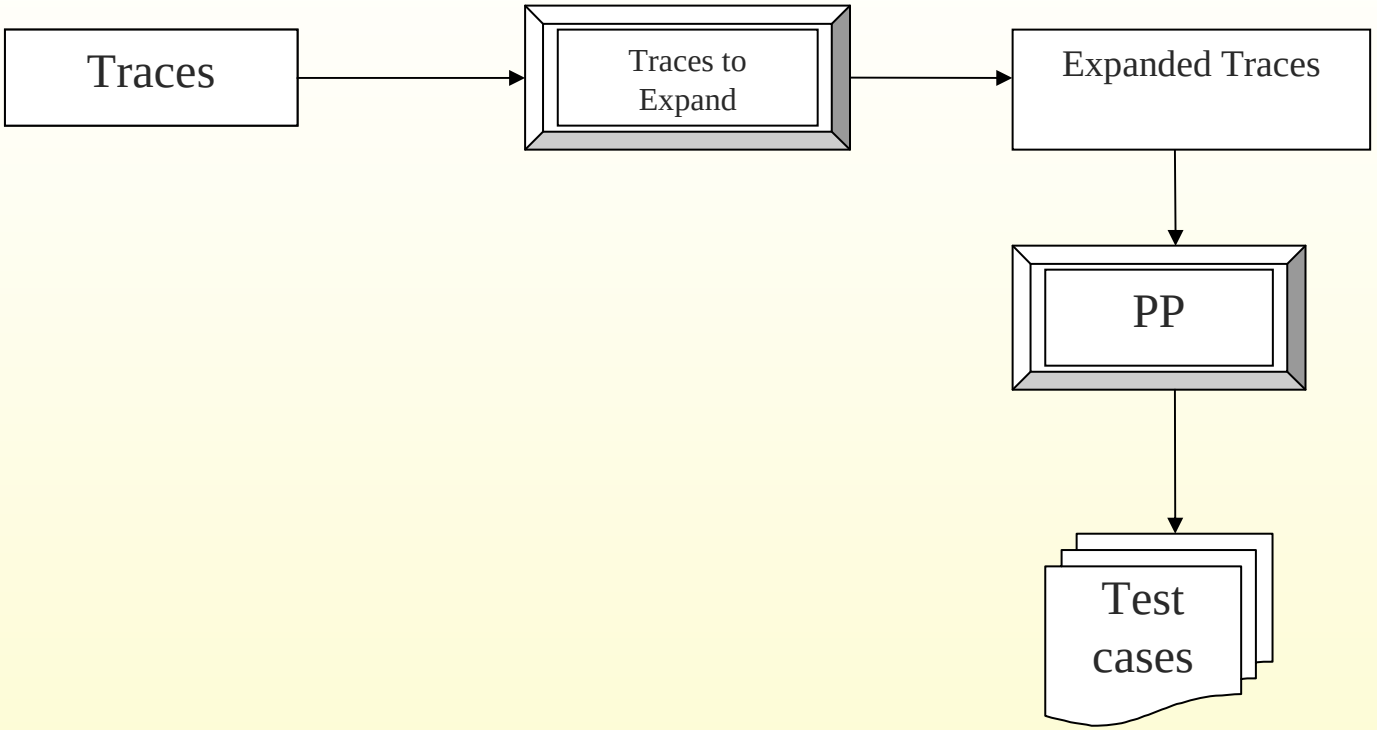
```
class Stack
  instance variables
  stack : seq of nat := [];

  operations
public Push : nat ==> ()
      ...
public Pop : () ==> nat
      ...
  traces
let x in set {1,5,10} in
Push(x); Pop()

end Stack
```

1. Push(1); Pop()
2. Push(5) ; Pop()
3. Push(10) ; Pop()

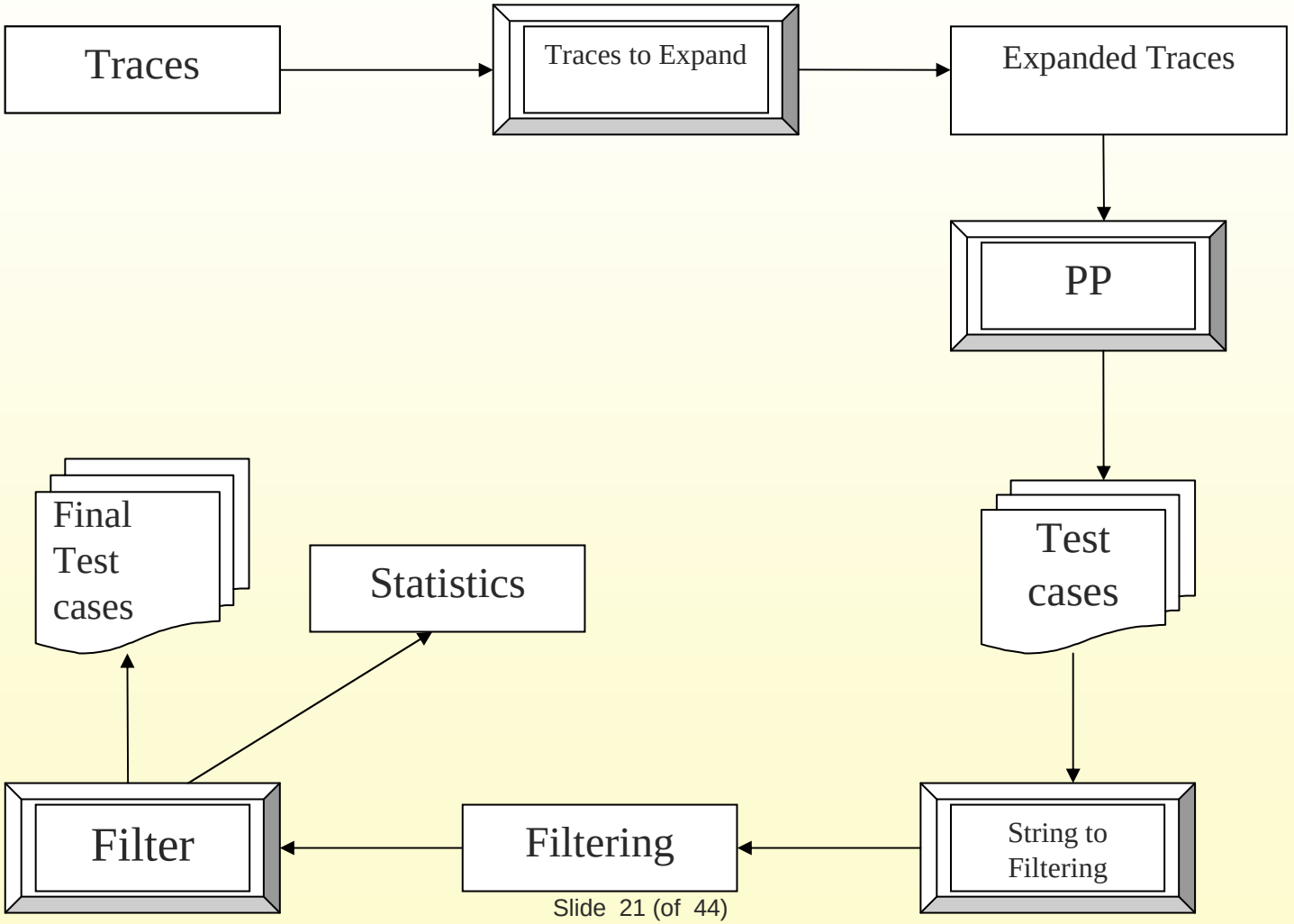
Combinatorial Testing schema



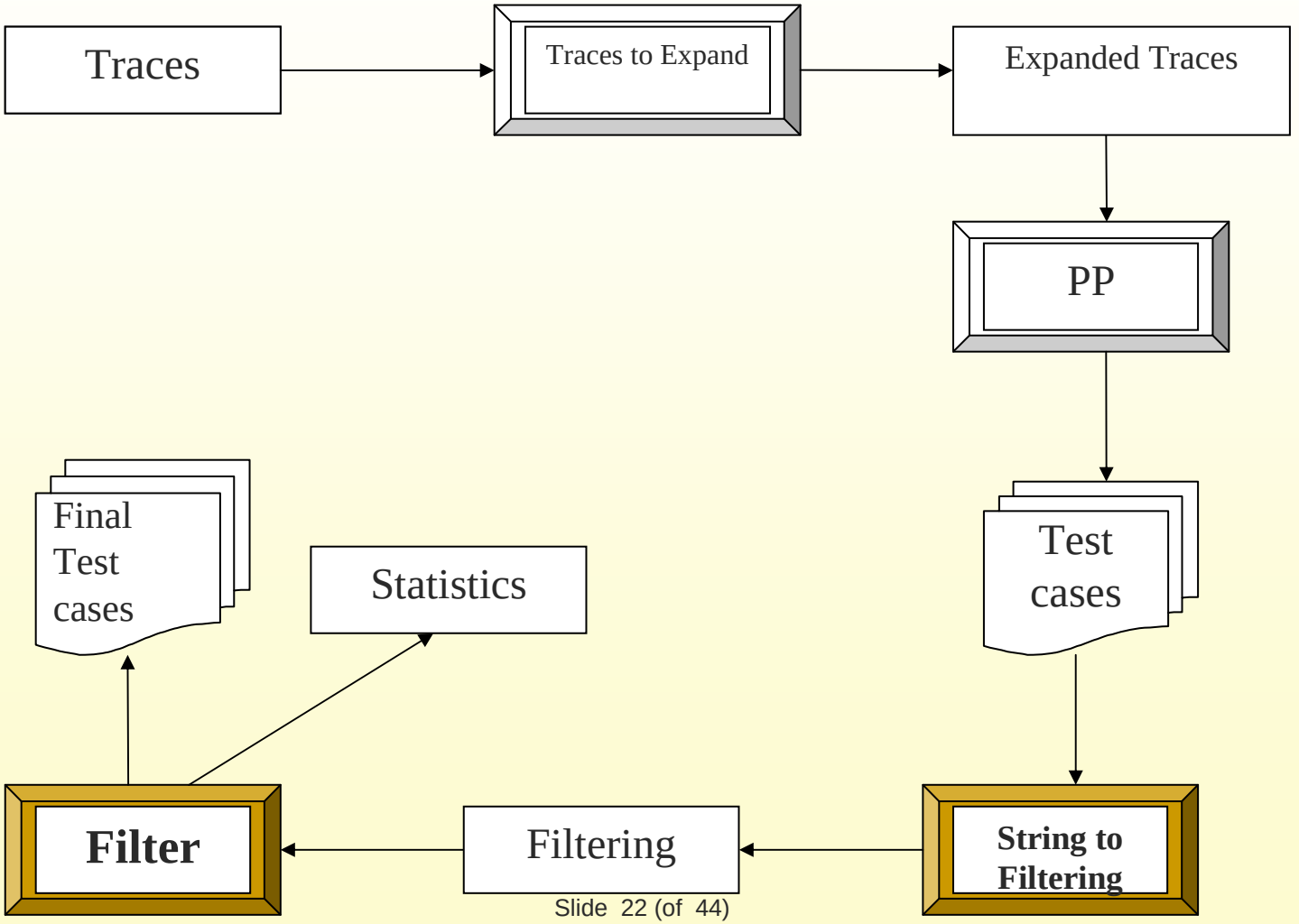
Combinatorial Testing schema



Combinatorial Testing schema



Combinatorial Testing schema



Filtering keywords

- Verdict of test cases
  - **PASS**
  - Execution failed
    - **FAIL** (output or operation)
    - **INCONCLUSIVE** (input parameters)

Filtering keywords

- Verdict of test cases
  - **PASS**
  - Execution failed
    - **INCONCLUSIVE** (input parameters)
    - **FAIL** (output or operation)

Filtering keywords

- Verdict of test cases
  - **PASS** Push(1)

```
 …
 operations
public
   Push : nat ==> ()
   Push(e) ==
      stack := [e] ^ stack
   pre e < 10
   post stack = [e] ^ stack~;
…
```

Filtering keywords

- Verdict of test cases
  - PASS
  - Execution failed
    - **INCONCLUSIVE** (input parameters) Push(11)

```
 ...
 operations
public
   Push : nat ==> ()
   Push(e) ==
      stack := [e] ^ stack
   pre e < 10
   post stack = [e] ^ stack~;
...
```

Filtering keywords

- Verdict of test cases
  - Execution failed: **FAIL** (output or operation)

```
class Stack
 instance variables
   stack : seq of nat := []
  inv len stack < 3;

 operations
public Push3 : nat ==> ()
Push3(e) ==
      stack := [e] ^ stack
post stack = [e] ^ stack~;

end Stack
```

Push3(1); Push3(1); Push3(1)

Filtering keywords

- Verdict of test cases
  - **PASS**
  - Execution failed
    - **FAIL** (output or operation)
    - **INCONCLUSIVE** (input parameters)

- **Prefix** of test case Push3(1); Pop():
  - Push3(1); Pop()
  - Push3(1); Pop()

Filter process

All/ Selected test cases:

{ Push(1) |-> not Tested,

Pop(); Push(1) |-> not Tested,

…,

test case n |-> not Tested}

Failed test cases:

{|->}

Filter process

All/ Selected test cases:

{ Push(1) |-> not Tested,

Pop(); Push(1) |-> not Tested,

…,

test case n |-> not Tested}

Failed test cases:

{|->}

- Did the prefix of “test case 1” fail?

Filter process

All/ Selected test cases:

{ Push(1) |-> not Tested,

Pop(); Push(1) |-> not Tested,

…,

test case n |-> not Tested}

Failed test cases:

{|->}

- Did the prefix of “test case 1” fail?

- Execute

  Did “test case 1” fail now?

Filter process

All/ Selected test cases:

{ Push(1) |-> PASS,

Pop(); Push(1) |-> not Tested,

…,

test case n |-> not Tested}

Failed test cases:

{|->}

- Did the prefix of “test case 1” fail?
- Execute
- Did “test case 1” fail now?

NO

NO

Filter process

All/ Selected test cases:

{ Pop() |-> FAILED,

Pop(); Push1() |-> not Tested,

…,

test case n |-> not Tested}

Failed test cases:

{Pop() |-> FAILED}

- Did the prefix of “test case 1” fail?

NO

- Execute

  Did “test case 1” fail now?

YES

Filter process

All/ Selected test cases:

{ Pop() |-> FAILED,

Pop(); Push(1) |-> not Tested

…,

test case n |-> not Tested}

Failed test cases:

{Pop() |-> FAILED}

- Did the prefix of “test case 2” fail?

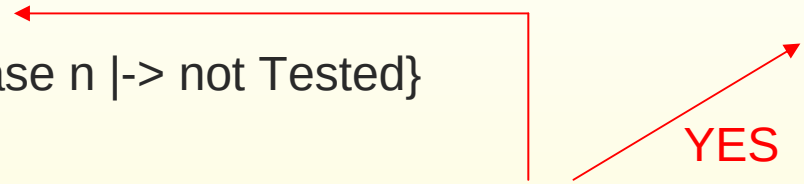
Filter process

All/ Selected test cases:

{ Pop() |-> FAILED,
…,
test case n |-> not Tested}

Failed test cases:

{test case 1 |-> FAILED,
Pop(); Push(1) |->FAILED}



- Did the prefix of “test case 2” fail?

Filter process

All/ Selected test cases:

{ test case 1 |-> FAILED,
…,
test case n |-> not Tested}

Failed test cases:

{test case 1 |-> FAILED,
test case 2 |-> FAILED}

- Did the prefix of “test case 2” fail?

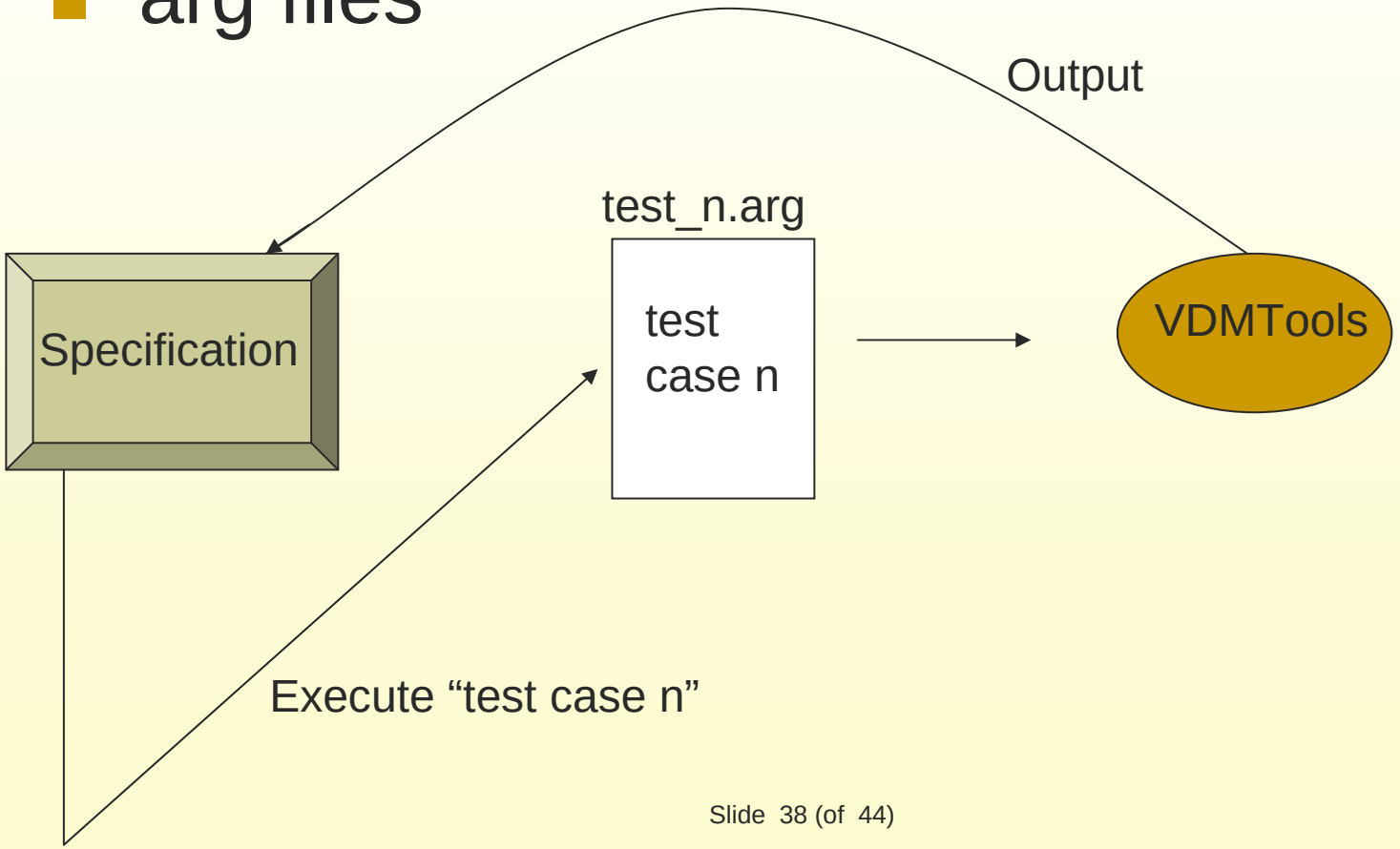
How is it possible to test a test case?

Test a test case

- arg files
  - sequence of VDM++ expressions separated by commas

Test a test case

- arg files



Output

Statistics

- Percentage/Total of failed test cases
- Percentage/Total of deleted test cases
- Percentage/Total of selected test cases

Full log

- All executed test cases
- Output from interpreter

Error file

- Test cases with a FAIL verdict
- Output from interpreter

Output

Statistics

- Percentage/Total of failed test cases
- Percentage/Total of deleted test cases
- Percentage/Total of selected test cases

Full log

- All executed test cases
- Output from interpreter

Error file

- Test cases with a FAIL verdict
- Output from interpreter

Output

Statistics

- Percentage/Total of failed test cases
- Percentage/Total of deleted test cases
- Percentage/Total of selected test cases

Full Log

- All executed test cases
- Output from interpreter

Error File

- Test cases with a FAIL verdict
- Output from interpreter

Reasoning

Pros

- More test cases
- Faster generation of test cases
- Faster analysis of output from interpreter

Reasoning

Pros

- More test cases
- Faster generation of test cases
- Faster analysis of output from interpreter

Cons

- Combinatorial explosion

Future work

- Continue implementing the combinatorial testing strategy
- Eclipse plugin