Using JML-based Code Generation to Enhance Test Automation for VDM Models

Peter W. V. Tran-Jørgensen[1] Peter Gorm Larsen[1]
Nick Battle[2]

AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

FUJITSU

14th Overture workshop, FM 2016
Limassol, Cyprus – November 7

Agenda

Introduction

Code generating traces

Performance results

Conclusion and future plans

Agenda

Introduction

Code generating traces

Performance results

Conclusion and future plans

Combinatorial testing (CT) for VDM

- CT is used to validate *VDM specifications*
  - Traces express (potentially large) test sets
  - Traces are *expanded* and *executed* automatically
  - Exhaustive testing of a model's *contracts*
  - Verdicts: `PASS`, `INCONCLUSIVE`, `FAIL`
- Traces are *interpreted*
- Subject to *combinatorial explosion*
  - Easy to construct traces that cannot be executed

Combinatorial testing (CT) for VDM

- CT is used to validate *VDM specifications*
  - Traces express (potentially large) test sets
  - Traces are *expanded* and *executed* automatically
  - Exhaustive testing of a model's *contracts*
  - Verdicts: `PASS`, `INCONCLUSIVE`, `FAIL`
- Traces are *interpreted*
- Subject to *combinatorial explosion*
  - Easy to construct traces that cannot be executed

Combinatorial testing (CT) for VDM

- CT is used to validate *VDM specifications*
  - Traces express (potentially large) test sets
  - Traces are *expanded* and *executed* automatically
  - Exhaustive testing of a model's *contracts*
  - Verdicts: `PASS`, `INCONCLUSIVE`, `FAIL`
- Traces are *interpreted*
- Subject to *combinatorial explosion*
  - Easy to construct traces that cannot be executed

Trace expansion

```
let x in set {1,2}
in
(
  ||(fun(x),op1(x)) | op2(x){1,2}
)
```

Trace expansion

```
let x in set {1,2}
in
(
  ||(fun(x),op1(x)) | op2(x){1,2}
)
```

```
x = 1; fun(x); op1(x);
x = 1; op1(x); fun(x);
```

Trace expansion

```
let x in set {1,2}
in
(
  ||(fun(x),op1(x)) | op2(x){1,2}
)
```

```
x = 1; fun(x); op1(x);
x = 1; op1(x); fun(x);
x = 1; op2(x);
x = 1; op2(x); op2(x);
```

Trace expansion

```
let x in set {1,2}
in
(
  ||(fun(x),op1(x)) | op2(x){1,2}
)
```

```
x = 1; fun(x); op1(x);
x = 1; op1(x); fun(x);
x = 1; op2(x);
x = 1; op2(x); op2(x);
x = 2; fun(x); op1(x);
...
x = 2; op2(x); op2(x);
```

Objectives

- Enable more exhaustive testing
  - Execute more tests
  - Reduce *memory consumption*
  - Reduce *execution time*
- Increase scope of CT
  - Support model implementation
  - Use CT to validate the model implementation

Objectives

- Enable more exhaustive testing
  - Execute more tests
  - Reduce *memory consumption*
  - Reduce *execution time*
- Increase scope of CT
  - Support model implementation
  - Use CT to validate the model implementation

Agenda

Introduction

Code generating traces

Performance results

Conclusion and future plans

The traces code generator

- Extension of Overture's JML translator
  - Contracts are translated to JML
- Traces are executed using *OpenJML*
- Traces are translated to Java
  - At runtime, traces form *object trees*
  - Tests are derived from the object tree
  - Expansion/execution is handled by a runtime library

The traces code generator

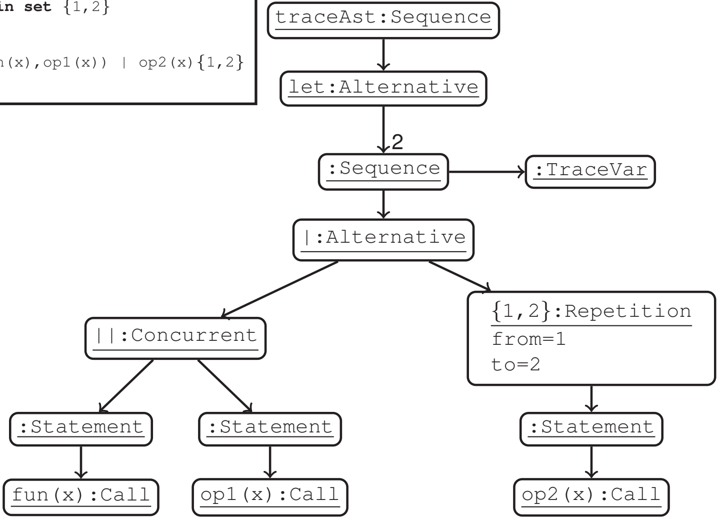
- Extension of Overture's JML translator
  - Contracts are translated to JML
- Traces are executed using *OpenJML*
- Traces are translated to Java
  - At runtime, traces form *object trees*
  - Tests are derived from the object tree
  - Expansion/execution is handled by a runtime library

The traces code generator

- Extension of Overture's JML translator
  - Contracts are translated to JML
- Traces are executed using *OpenJML*
- Traces are translated to Java
  - At runtime, traces form *object trees*
  - Tests are derived from the object tree
  - Expansion/execution is handled by a runtime library

Traces at runtime

```
let x in set {1,2}
in
(
 ||(fun(x),op1(x)) | op2(x){1,2}
)
```



The call statement (a leaf)

```
Call callStm_3 = new Call() {
  public Boolean isTypeCorrect() {
    try {
      //@ assert Utils.is_nat(x);
    } catch (AssertionError e) {
      return false;
    }
    return true;
  }
  public Boolean meetsPreCond() {
    return pre_op2(x);
  }
  public Object execute() {
    return op2(x);
  }
  public String toString() { ... }
};
```

Agenda

Introduction

Code generating traces

Performance results

Conclusion and future plans

Execution results

Size	VDMJ-3.1.1 [ms]	Overture-2.3.2 extension [ms]	Code Generated [ms]
1	46	124	211
2	465	621	633
3	2,139	3,288	3,217
4	8,692	9,068	29,032
5	35,610	57,999	279,401
6	379,635	failed	2,953,318

- Overture fails to run the tests (runs out of memory)
- VDMJ completes the tests in $\underline{\approx 6.3}$ minutes
- Code generated traces take $\underline{\approx 49.2}$ minutes
- The code generator uses Overture's expansion
- VDMJ uses more efficient expansion

Analysing the results

- Execute trace as plain Java program (No OpenJML)
  - Execution time: $\approx$ 34 seconds
  - No constraints are checked
  - Would be faster with VDMJ's expansion algorithm
- Remove JML, execute using OpenJML
  - Execution time: $\approx$ 11.2 minutes
  - Would expect it to approach 34 seconds
- *Unexpectedly large overhead of using OpenJML*

Analysing the results

- Execute trace as plain Java program (No OpenJML)
  - Execution time: $\approx$ 34 seconds
  - No constraints are checked
  - Would be faster with VDMJ's expansion algorithm
- Remove JML, execute using OpenJML
  - Execution time: $\approx$ 11.2 minutes
  - Would expect it to approach 34 seconds
- *Unexpectedly large overhead of using OpenJML*

Analysing the results

- Execute trace as plain Java program (No OpenJML)
  - Execution time: $\approx$ 34 seconds
  - No constraints are checked
  - Would be faster with VDMJ's expansion algorithm
- Remove JML, execute using OpenJML
  - Execution time: $\approx$ 11.2 minutes
  - Would expect it to approach 34 seconds
- *Unexpectedly large overhead of using OpenJML*

Agenda

Introduction

Code generating traces

Performance results

Conclusion and future plans

Conclusion

✓ Technique to code generate traces
- Main contribution of this work
- Can be implemented using other technologies

✗ Currently the performance is poor
- Contradicts the expectation
- OpenJML is the bottleneck

Future plans

- Very recent releases of OpenJML
  - Recent activity, V0.8.1 (November 2, 2016)
  - Java 8 support
  - Re-run experiments using newest OpenJML
- Addressing the performance issues
  - Use more efficient expansion algorithm (VDMJ)
  - Investigate use of other contract-based technologies
    - Microsoft Code Contracts (ongoing work)

Future plans

- Very recent releases of OpenJML
  - Recent activity, V0.8.1 (November 2, 2016)
  - Java 8 support
  - Re-run experiments using newest OpenJML
- Addressing the performance issues
  - Use more efficient expansion algorithm (VDMJ)
  - Investigate use of other contract-based technologies
    - Microsoft Code Contracts (ongoing work)