Trace-Directed Modelling
Mid-Project Meeting Report

Timothy C. Lethbridge

CRuiSE (Complexity Reduction in Software Engineering) Research Group

University of Ottawa

Dec 8, 2010

http://www.site.uottawa.ca/~tcl

Team members and what they are doing

All team members

- Spent a lot of time learning LTTng, code generation framework, Papyrus tool

Sultan Eid

- Masters
- Key initial task: Trace case specification and code generation from models

Hamoud Aljamaan

- PhD
- Successfully completed comprehensive exam
- Key tasks:
  - Usability of trace specifications
  - Making trace results visible in model

Some team members on other projects are supporting the work

Key recent progress

Draft specification of a language for specifying tracing in a UML model

- UML already has several 'add on' languages
  - OCL
  - ALF Action language (under development)
- We are adding another

How would this be used?

- Either
  - Direct the code generator to inject trace code ready for later activation
  - Create a separate trace application that can instrument a system already installed

The following slides represent a proposal that can be easily changed

Discussion is welcome

We are not yet committed to any particular

- Syntax
- Semantics
- Architecture

uOttawa

Prior research we built on: TraceSQL Declarative Tracepoints

SQL-like language for writing trace-injection applications

- Dynamically instrument the target system when loaded
- Borrows concepts from aspect-oriented technology

Reference

- Q. Cao, T. Abdelzaher, J. Stankovic, K. Whitehouse, L. Luo, "Declarative Tracepoints: A Programmable and Application Independent Debugging System for Wireless Sensor Networks", SenSys'08

General TraceSQL syntax

- TRACE {...} FROM {...} EXECUTE {...} WHERE {...}

TraceSQL Declarative Tracepoints 2

Examples

```
INTEGER @numYields = 0;
TRACE yield() FROM syscall.c EXECUTE {
  @num_yields++;
}
WHERE {
  READ msend->lock FROM radio.c == 1
}

// Write the number of yields in 60s periods 10 times (i.e. 10 mins)
TRACE PERIOD 60s FOR 10 EXECUTE {
  RECORD @numYields;
  @numYelds = 0;
}
```

Weaknesses of TraceSQL

- **Not object-oriented (I.e. UML/C++ compatible)**
- **Over-specialized for embedded sensor device operating systems like LiteOS**
- **Not open source apparently**
- **Ugly syntax with all-caps**
  - I don't think SQL is a good basis

How we propose to adapt TraceSQL concepts for model-oriented tracing

The from clause is replaced by placing trace statements inside descriptions of the matching model elements

```
class X {
  trace method1();
  execute {
    record("method1 called");
  }
```

Simplified syntax for tracing in classes

```
class [classpattern] {
  trace {[traceItem]* } | traceItem
  [execute {[executeItem]* } | executeItem]
  [where {[precondition]* }]?
}
```

Tracing in state machines also available

General architecture for model-level tracing specifications

Operates in the context of full model-driven development

- Generation of the system from models
- Models have classes, state machines
- ‘Action Language’ C++ methods can be interspersed

UML model elements enhanced with trace specifications

- These can be written at design time and maintained in a library for later use
  - Can be activated at run time
- Alternatively, when debugging a system, go to the model and specify new model-level tracing

Code generator inserts tracepoints compatible with GDB, UST etc.

At run time GDB / UST tracepoints execute what is specified

- Output tagged with model element IDs from the original model, to allow analysis at the model level

Our specific approach and architecture

- **In process of making open source in GoogleCode under MIT License**
- **We are writing everything in**
  - JET templates for code generation
  - Model driven tools UML/Umple for everything else
    - Because
      - It speeds our work
      - We want to ‘eat our own dogfood’
    - But Umple is just a tool that generates pure Java
      - So if later users don’t want to use Umple, they don’t need to
- **Thorough test-driven development**
- **Runs on command line or plugs into any Eclipse-based modeling tool**
  - Generates Papyrus XMI
  - Full integration with Papyrus will be done later

Details of model-level tracing syntax: Basic tracing of a method in class X

When m1() is called, output "X" into the trace, along with tags indicating the model class X and m1()

```
trace m1() execute "x";
```

Or

```
trace m1() execute record("x");
```

Or

```
trace m1() execute {record("x");}
```

In general the {} can be left out unless there are multiple items

Can leave out 'execute' to just get a record of the name of the item traced

```
trace m1();
```

Multiple trace items and conditional tracing

When m2 or m3 called, print x and the result of the method

```
trace {m2(); m3();} execute {record("x",result);}
```

When m4 called, print y provided the where condition is true

– Note that where clause statements represent *preconditions*

```
trace m4() execute "y" where attr7>5;
```

Expanding and limiting what is traced

Pattern matching

trace m*() …

Tracing when a certain value is returned by a method

trace m5()<5 …

Tracing method exit only (otherwise traces entry+exit)

trace exit m6() …

Tracing method entry only

trace entry m6()

Tracing other things only in the control flow (between entry and exit of a method)

trace cflow m7() {class Y {trace m8(); m9();}}

Tracing when attr1 changes

trace attr1 …

Or

trace setAttr1() …

More attribute tracing

Trace any time attr2 is set to a value exceeding 5

```
trace attr2 > 5 …
```

Trace any sets of attr3 to value 7

```
trace attr3 == 7 …
```

Tracing any time an attribute is accessed

```
trace getAttr4() …
```

Tracing associations

Trace any changes to association assoc1

```
trace assoc1 …
```

Trace changes to assoc1 such that the cardinality becomes 0

```
trace cardinality(assoc1) == 0 …
```

Tracing based on time or occurrences

Trace the first 100 changes to an attribute

trace attr8 for 100 …

- Afterwards, this trace directive is ignored

Print out attr3 every 30ms

trace period 30ms execute attr3

Trace changes to attr4 for 12ms

trace attr4 during 12ms …

- Afterwards, this trace directive is ignored

Trace until a condition becomes true

Trace changes to attr5 until attr6 is set to a value > 3

– even if the condition becomes false again afterwards

```
trace attr5 until attr6>3 …
```

The above can be combined

– Trace up to 100 calls to method1, but stop tracing if it returns a value less than zero

```
trace method1() for 100 until method1()<0
```

Named trace cases

You can name a set of tracing rules

- For activating:
  - At a specific *point in time*
  - When a certain *condition becomes true*

- And deactivating

Conceptually, the previous slides referred to a default <u>unnamed</u> trace case

- e.g. initially loaded

Named tracecase declarations

The same name appearing in multiple model entities adds to the trace case

– This is ‘mixin’ capability

```
tracecase tc1 {
  trace attr6 execute “a6”
}

tracecase tc1 {
  trace attr7 execute {“a7”;  count++;}
}

tracecase tc1 {
  trace attr6 execute count--;
}
```

Tracecases can have local attributes

They are accessible inside it and local to each specific activation

```
tracecase tc2 {
  Integer i;
  String s;
}
```

Execte clause actions: Recording output

Any list of expressions or single expression can follow the record keyword

– Generates code that causes LTTng or UST to output CTF-compatible data

Record a constant

```
record "constant";
```

Record the value of an attribute

```
record attr1;
```

- The record keyword can be omitted above for single items

Record several things

```
record("Got here", attr1, attr4)
```

Execute clause actions for activation

Activation of a trace case

activate tc1

Activate a trace case in the context the instance that matched the trace clause

activate tc3 on this

- Until you do the above, tracing would have been done on all objects of a given class

Activate a trace case in the context of the current thread

activate tc4 on thisThread

- Without this, tracing is done in all threads

More activation controls

Activate a trace case in the context of a particular object or set of objects

activate tc5 on assoc3

Deactivate a trace case in all contexts

deactivate tc2

Activate a trace case for a period of time

activate tc3 for 1s

More execute clause statements

activate a trace case until a condition becomes true

activate tc4 until attr6>4

Combining various elements

activate tc5 on this during 12ms

Set an attribute

– Modifies the functioning of the base system

attr7 = 5

Tracing transitions in a state machine

Example where a state machine is embedded in a class

```
class c1 {
  sm1 {
    // trace all occurrences of ev1 that effect the state
    trace ev1 execute “ev1”;
    state1 {
      // trace something only when in state1
      trace attr3 …

      // trace a particular transition here by specifying the event
      trace ev2 …
    }  // end of state
  } // end of state machine
}  // end of enclosing class
```

More trace machine tracing examples

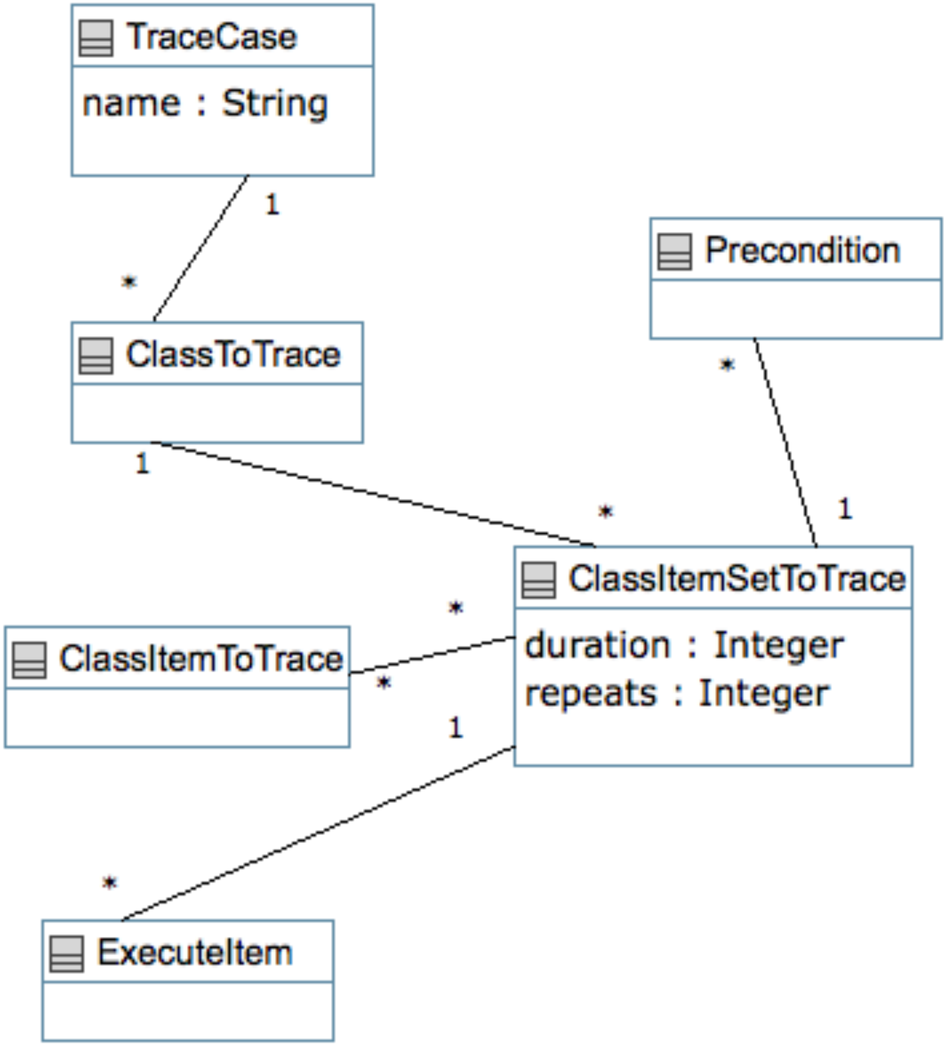
Tracing any change of state

```
class c1 {trace sm1;}
```

Tracing a pure state machine that can be plugged into any class

```
statemachine sm2 {
  // trace ev3 in this state machine
  trace ev3;
  trace entry;
  statea {
    trace {attr3; ev5; m6();} execute {activate tc7;}
    trace exit;
  }
}
```

Partial Specification-Time Metamodel for Tracing



uOttawa

Directions in this aspect of the research

Mostly conceptual so far

- Although the parsing and code generation infrastructure is primed for adding the trace language

Step 1: Review the above with stakeholders to refine

Step 2: Prototype it

- Use test-driven development
  - Complete parser
  - Inject code in code generator
- Hope to have a concrete demo by mid-year meeting

Other activities:

- Render trace results back into the model

Longer-range objectives

Conduct empirical studies

- Usability of the language

Try on significantly sized UML models

Reverse engineer real systems to models

- Then trace systems using our approach

Other ideas to extend and integrate with other subprojects

Integrate with live tracing

- Abstract results to an instance-level view in the modeling tool

Integrate with trace abstraction work

- Abstract the traces back to models