Overview of the SMUML Toolset

Heikki Tauriainen
Helsinki University of Technology (TKK)
Laboratory for Theoretical Computer Science

May 16, 2008

Abstract

This document gives a brief introduction to the main features of the SMUML toolset as they are presented in the graphical user interface of the toolset's front-end script. The document also provides a reference to the on-line documentation of the individual tools in the toolset.

Contents

1 Introduction

This document provides a brief summary of the main features available in the SMUML toolset, focusing on their use via a front-end graphical user interface (GUI). It is assumed that the reader is familiar with the Unified Modeling Language (UML) 1.4 subset used within the toolset (for more information, see the separate document *The SMUML UML subset* included in the toolset documentation) as well as constructing UML 1.4 models as XML Metadata Interchange (XMI) 2.0 files (using, e.g., the Coral metamodeling tool).

1.1 Acknowledgments

The SMUML toolset was designed by Jori Dubrovin, Tommi Junttila, Toni Jussila, Timo Latvala, Sami Liedes, Ilkka Niemelä, Vesa Ojala, Juhani Peltonen and Heikki Tauriainen at Helsinki University of Technology (TKK), Laboratory of Theoretical Computer Science. The financial support of the Finnish Funding Agency for Technology and Innovation (TEKES), Nokia, Conformiq, and Mipro is gratefully acknowledged.

2 Prerequisites

The front-end to the SMUML toolset requires the following software (not included in the SMUML toolset distribution) to be installed:

- A Python interpreter (at least version 2.4 or later).[1]
- The Coral metamodeling tool (and its Python application programming interface).[2]
- The GTK+ graphical user interface toolkit libraries, including the PyGTK application programming interface (version 2.8 or later recommended).[3] The tools in the SMUML toolset can be used also in text mode, in which case these libraries are not required.

Additionally, at least one of the following back-end tools must have been installed before any models can be analyzed:

- The NuSMV model checker (with satisfiability solvers required for bounded model checking).[4]
- The Spin model checker[5] and a C compiler.

[1] http://www.python.org/
[2] http://mde.abo.fi/confluence/display/CRL/
[3] http://www.gtk.org/, http://www.pygtk.org/
[4] http://nusmv.irst.itc.it/
[5] http://spinroot.com/

- A Satisfiability Modulo Theories (SMT) solver. The supported SMT solvers are those accessible via the PySMT application programming interface (included in the SMUML toolset distribution); currently, support is provided for the solvers CVC3[6], MathSAT[7], STP[8], Yices[9] and Z3[10].

Before running any tools in the toolset, check that your `PYTHONPATH` search path includes the locations of the Python API for the Coral metamodeling tool, and the PyGTK API (if you intend to use the tools via their graphical user interface).

3 Features of the Graphical Front-End

To start the front-end in graphical mode, type

[SMUML_root]`/bin/analyze.py --gui`

at the command prompt (where [SMUML_root] is the name of the root directory of the SMUML toolset). Provided that the libraries required by the toolset have been installed correctly, you should be presented with the window similar to the one shown in Figure 1. This window consists of a text entry field for the name of an XMI 2.0 project file containing a UML 1.4 model to be analyzed, a button for opening a file chooser dialog to select a project file, and the following tabs:

Preprocessing The *Preprocessing* tab contains options which can be used to select transformations to be applied to the model before analysis. See Section 3.2.

Model checking The *Model checking* tab contains options for choosing the back-end to use for model checking, and options for controlling the behavior of the selected model checking back-end. See Section 3.3.

Abstraction refinement The options on the *Abstraction refinement* tab control the behavior of counterexample-based automatic data abstraction refinement. See Section 3.4.

Counterexamples The *Counterexamples* tab provides some post-processing options for counterexamples found in model checking. See Section 3.5.

Configuration The *Configuration* tab can be used to specify paths for scripts and external tools needed by the front-end script or the model checking back-ends. See Section 3.1.

About The *About* tab displays information about the SMUML toolset.

Additionally, the window contains the buttons *Start* and *Quit* for starting model analysis and exiting the program, respectively.

3.1 Configuring Paths for Scripts

Before starting model analysis for the first time, you should check that the paths of external scripts and tools specified on the *Configuration* tab

[6] `http://www.cs.nyu.edu/acsys/cvc3/`
[7] `http://mathsat.itc.it/`
[8] `http://theory.stanford.edu/~vganesh/stp.html`
[9] `http://yices.csl.sri.com/`
[10] `http://research.microsoft.com/projects/z3/`

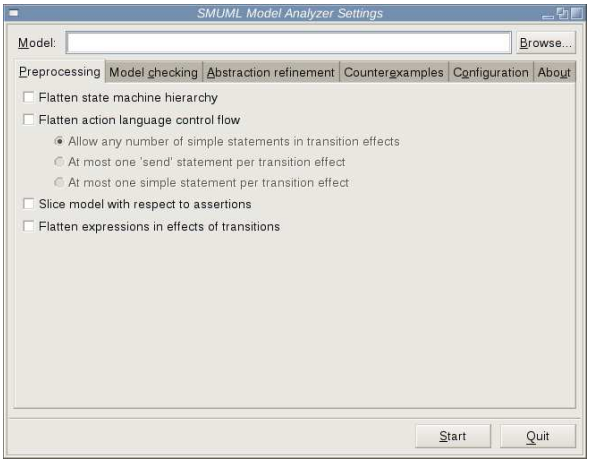

Figure 1: Graphical user interface for changing model analysis settings.

are appropriate for your environment. By default, this tab displays the contents of the textual configuration file `paths.conf` in the current working directory (if such a file exists); an alternate configuration file can be loaded by clicking the *Browse...* button on this tab. A configuration file template is included in the SMUML toolset distribution.

The configuration file for the paths of external scripts and tools should contain zero or more lines of the form

[key]=[value]

Empty lines and the part of each line following the first `#` are ignored. Each [key] represents the name of a variable (a script or an external tool), and each [value] is a value for this variable (the command to run the script or tool with possible command line arguments). If a value begins or ends with white space, enclose the value in single or double quotes. Previously defined variables (including variables defined in the environment) can be referenced using the syntax `$`[key] or `${`[key]`}`.

The easiest way to prepare a configuration file is to modify the sample configuration file included in the SMUML toolset distribution to your needs. In particular, check that the keys `PYTHON` and `SMUML`, the key `nusmv` specifying a path for the NuSMV model checker, the keys for paths for the supported SMT solvers (`cvc3`, `mathsat`, `stp`, `yices`, `z3`), and the keys needed for using the Spin model checker (`spin`, `CC`) have appropriate values if you intend to use these model checkers or solvers.

3.2 Model Preprocessing Options

The *Preprocessing* tab contains options for choosing transformations to be applied to the model before model checking. These transformations are sometimes needed to normalize models before they can be analyzed since the model checking back-ends and abstraction refinement tools have varying constraints for models given to the tools as input. To apply a particular transformation, check its associated checkbox on this tab. The available transformations are:

Flatten state machine hierarchy Replace all hierarchical state machines in the model with state machines containing no composite states other than their top state. Additionally, ensure that all transitions leaving a state of any of these state machines are either triggered by a signal, or they are all completion transitions.

Flatten action language control flow Make the control flow in all action language statements involving a branch (such as conditional statements, or `for` and `while` loops) explicit in the transition structure of the state machines such that all effects of transitions in the resulting state machines contain only statements which will be executed sequentially (empty statements, assignments, and `assert` and `send` statements). Note that every sequence of these statements occurring within the effect of a single transition will be executed atomically; to control the atomicity, the sequences of statements can be split further into segments, each of which contains either (i) at most `send` statement, or (ii) at most one simple statement.

Slice model with respect to assertions Perform a static analysis on each state machine in the model to find action language statements whose execution will not affect the value of any variable in the condition of any assertion statement in the model; replace all such statements with empty statements to try to reduce the effort needed for model checking.

Flatten expressions in effects of transitions Augment classes in the model with new temporary variables to be used for evaluating compound arithmetic subexpressions in the effects of transitions in the state machines in the model. (Guard expressions of transitions will be left unchanged.) For example, the compound arithmetic expression

```
((y + z) - w) * q
```

could be transformed to a plain reference to a temporary variable `tmp_2` initialized using the sequence of assignments

```
tmp_0 = (y + z); tmp_1 = tmp_0 - w; tmp_2 = tmp_1 * q;
```

This transformation may reduce the effort needed for inlining definitions of abstract operations when using automatic abstraction refinement.

The selected model transformations will always be applied in the order from top to bottom.

Table 1 shows a high level overview of the constraints on input models for various model transformation and analysis tools included in the toolset. Input models for tools which do not support hierarchical state machines need to be first processed with the state machine hierarchy flattener if necessary; similarly, input models containing complex action language control flow constructs need to be processed through the action language flattener before they can be analyzed. Note that some unsupported constructs (for example, action language expressions which contain arithmetic expressions having side effects, such as pre- or postfix increment operations) cannot be removed from the models automatically using tools currently available in the SMUML toolset.

Table 1: Summary of input model features supported by model transformation and analysis tools. A ✓means that the model feature is supported directly by the tool.

	Model slicing	Expression flattening	Uboco	Suboco	Proco	Abstraction refinement	Counterexample simulation
hierarchical state machines, completion transitions mixed with triggered ones[a]			✓	✓	✓		✓
action language constructs with control flow branching[b]							
more than one `send` statement per transition[b]	✓	✓			✓	✓	✓
arithmetic expressions with side effects[c]							

[a] can be removed by flattening state machine hierarchy
[b] can be removed by flattening action language control flow
[c] cannot be removed automatically using tools in the toolset

3.3 Model Checking Options

The *Model checking* tab contains options which control how a model is to be analyzed. The SMUML toolset depends on external back-end model checkers (or Satisfiability Modulo Theories solvers) in the actual model analysis phase: for each of these back-ends, the SMUML toolset includes an interface via which the model to be analyzed is translated into input formalisms supported by the external tools. The supported interfaces and back-ends are:

Uboco This interface translates models into the input language of the NuSMV symbolic model checker. This model checking back-end supports model analysis using bounded model checking (BMC) with the help of a satisfiability (SAT) solver, or exhaustive model checking using binary decision diagrams (BDDs). See *Uboco User's Guide* for more information.

Suboco This interface supports model analysis via bounded model checking by translating the model checking problem into Satisfiability Modulo Theories (SMT) problem instances, which are then solved using any of the SMT solvers supported by the toolset. For more information, see *Suboco User's Guide*.

Proco This interface translates models into Promela, the input language of the Spin explicit state model checker. The model checking is then done by a verifier generated from this model by Spin and a C compiler. For more information, see *User's guide to* proco *version 2.0*.

The SMUML toolset can be used to analyze models for the following errors:

Assertion violations Check whether the model has a computation which leads to a violation of the condition of an `assert` statement.

Presence of deadlocks Check whether the model has a computation which ends in a state in which no further progress is possible. (Not compatible with abstract models.)

Implicit consumption of events Check whether the model has a computation in which some signal event is consumed without triggering a transition.

Runtime errors (Uboco and Suboco; for Proco, runtime error checking is always enabled) Check whether the model has a computation which leads to a runtime error (e.g., division by zero).

Event queue overflows (Proco) Check whether the model has an execution where the maximum capacity of the event queue of some object is exceeded.

Additionally, there are a number of options specific to each model checking back-end. Most of these settings, which are adjusted using command line options when using the back-end interfaces as stand-alone tools via the console, are collected in the *Additional options* column for each back-end; see the documentation for each back-end interface for more information. The *Extra options* field can be used to specify options (from the command line options supported by the back-end) which do not have a corresponding setting in the GUI.

3.4 Abstraction Refinement Options

The *Abstraction refinement* tab can be used to enable automatic counterexample-based data abstraction refinement, and choose settings related to generating operations for abstract data types. Automatic abstraction refinement is intended to be applied always directly to concrete models; automatic abstraction refinement is not supported (at least in the way that might be expected) for abstract models generated with the model abstraction tools using user-defined type libraries.

3.4.1 Overview of Data Abstraction Refinement

The purpose of data abstraction is to reduce the state space of a model to be analyzed by assigning its attributes abstract types with domains that are smaller than the domains associated with the original types of the attributes. Each abstract type has also operations whose results over-approximate the results of the corresponding concrete operations. An abstract model can be generated via a source-to-source transformation by replacing each concrete operation used in an action language expression in the model with the definition of the corresponding abstract operation for some abstract type.

The SMUML toolset supports data abstraction of integer attributes. For automatic abstraction refinement, abstract types for the attributes are chosen from a family of partitions of the set of integers into intervals. Each partition in this family is characterized by a finite set of integers to be separated from the partition: the domain of the abstract type corresponding to the partition then consists of these integers and the intervals

between them. Automatic abstraction refinement starts with abstracting all integer attributes in the model using the trivial partition with no elements separated from it; abstraction refinement then proceeds by analyzing counterexamples found in model checking to gradually refine the partitions assigned to the model's attributes until the types of the attributes are precise enough to show the existence of a computation which leads to an error in the original unabstracted model, or prove that the model has no such computations (of at most some given length when using bounded model checking).

3.4.2 Settings

The abstract models used in the SMUML toolset are generated using a source-to-source transformation as described above. The definitions of abstract operations are generated automatically using a Satisfiability Modulo Theories solver. The toolset provides several choices for the integer semantics to use for generating abstract types:

Bounded integer semantics When using this semantics, the automatically generated abstract operations will approximate the semantics of the corresponding concrete operations of the `int` type of the Jumbala action language (32-bit signed integers which wrap on arithmetic overflow). The approximations are safe overapproximations of the concrete operations under the assumption that no runtime errors (e.g., division by zero) occur. This option provides the most faithful approximation of the actual semantics of the Jumbala action language and has no limitations as to which operations can be used in action language expressions contained in a model—that is, all arithmetic (`+`, `-`, `*`, `/`, `%`, `&`, `|`, `~`, `^`, `<<`, `>>`, `>>>`) and test (`==`, `!=`, `<`, `<=`, `>`, `>=`) operators of the Jumbala action language are supported. (The Boolean operators `&&`, `||` `!` and the ternary conditional operator `?:` are also supported; these operators are never actually abstracted, however.)

Unbounded integer semantics Choosing this semantics for integers allows abstract operations to be generated by exploiting the properties of integer partitions, which is usually faster than generating the operations using a generic SMT solver. Although assuming the underlying concrete operations to operate on unbounded integers will in general produce semantically unsound approximations for operations in the action language, this semantics can still be useful if there is no danger of (or need to prepare for) arithmetic overflow. When using this semantics, however, abstraction is supported only for the basic arithmetic (`+`, `-`, `*`, `/`) and test (`==`, `!=`, `<`, `<=`, `>`, `>=`) operators.

Mixed integer semantics When using this option, abstract operations will be generated using a combination of the previous two semantics: basic arithmetic (`+`, `-`, `*`, `/`) and test (`==`, `!=`, `<`, `<=`, `>`, `>=`) operations (as well as mappings between values of abstract types) will be generated by approximating the corresponding operations on unbounded integers, and the remaining arithmetic operators (`%`, `&`, `|`, `~`, `^`, `<<`, `>>`, `>>>`) will be generated by approximating operations on bounded integers.

Abstract operations can be generated automatically using any of the Satisfiability Modulo Theories solvers supported by the SMUML toolset.

Finally, the *Limit for flattening abstract operations* controls how eagerly the abstract model generator should apply partial evaluation ("flattening") to compound action language expressions which contain only references to variables of abstract (or Boolean) type. For example, instead of first replacing the subexpression `(a + b)` in the compound expression `(a + b) - c` with the definition of the addition operation for some abstract type (resulting in an expression `e`) and then (in effect) the expression `e - c` with the definition of the subtraction operation for an abstract type, the entire expression could be replaced with a case analysis enumerating the possible combinations of values for the expression's "input attributes" `a`, `b` and `c` and the value of the expression under each combination of values. This transformation may help to reduce the size of expressions in an abstract model, but on the other hand may make abstract models converge more rapidly towards the concrete model in abstraction refinement. The value for the limit is used to restrict flattening to those expressions, the size of whose input domain (i.e., the Cartesian product of the domains of its input attributes) is at most equal to this limit. The value `None` removes this upper limit, enabling unconditional flattening of all abstract operations.

3.5 Counterexamples

The *Counterexamples* tab contains a text entry for specifying the name of a file in which to save a counterexample trace leading to an error in the model (if a real counterexample is found in model checking), and whether to also simulate the counterexample after model checking. Counterexample traces are saved in the SMUML toolset's generic counterexample format. Traces in this format can be simulated independently using the trace simulator included in the toolset.

3.6 Running the Analysis

After setting model analysis options, the analysis can be started by clicking the *Start* button. During the analysis, the graphical user interface displays some debugging information about the progress of the analysis. After the analysis is complete, a log of these debug messages can be saved to a file by clicking the *Save log as...* button. Clicking the *Exit* button returns to the model analysis settings dialog.

4 Known Limitations

In addition to the restrictions on input models supported by the individual tools in the SMUML toolset (Table 1), the toolset has the following known limitations:

- Input models generated using Uboco may contain features which are supported by the NuSMV model checker only in bounded model checking mode.
- The extent of support for different Satisfiability Modulo Theories solvers may vary (currently only partial support for some solvers is available). The Yices SMT solver (version 1.0.9) is known to work with the tools in the SMUML toolset without severe limitations.

- The Proco model checking back-end does not support the analysis of abstracted models. Consequently, the back-end cannot be used for abstraction refinement, either.
- Checking for the existence of deadlocks in abstracted models is not recommended since model abstraction may introduce spurious deadlocks which are normally forbidden by UML semantics (which may, for example, cause deadlocking counterexample traces from abstract models to be unsimulatable, thus preventing, e.g., counterexample analysis for abstraction refinement).
- Checking for runtime errors is not supported for abstracted arithmetic operations (enabling runtime error checking will have no effect).
- The graphical user interface to the front-end does not provide a method to interrupt the model analysis. (Killing an external model checker process may in some cases be used as a workaround.)
- The graphical user interface may seem to freeze in the "Generating abstract model" phase when saving an abstract model file. This is normal.
- Using a large value (or the value `None`) for the limit for flattening abstract expressions for abstraction refinement may cause the Python interpreter to exceed its default maximum recursion depth.
- Depending on the types of error conditions selected to be checked, counterexample trace simulation may not always be possible up to the first error of one of the selected types if another error (of any type) occurs already before this error in the trace.

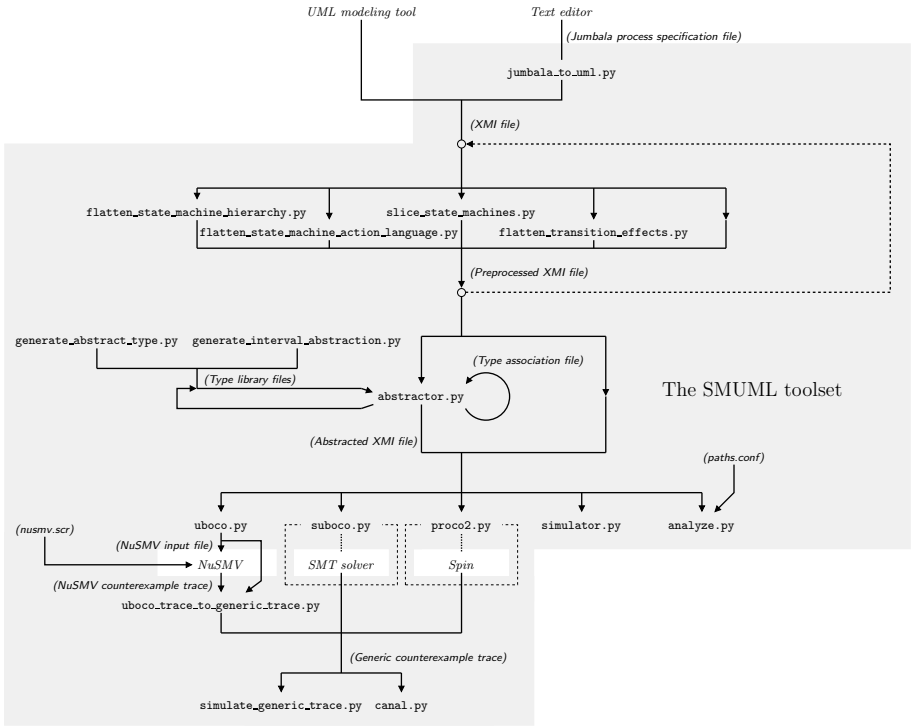

Figure 2: Overview of the workflow using tools in the SMUML toolset. The extent of the SMUML toolset is displayed in a grey background.

A List of Tools in the SMUML Toolset

Most tools included in the SMUML toolset can be used also independently via the console using the Python interpreter. This appendix collects the on-line documentation of the main tools in the SMUML toolset for reference. Tools related w.r.t. their functionality are grouped by section. Figure 2 gives an overview of the dependencies between tools in the toolset (see also Table 1 and Section 4 for more details on the requirements and limitations of tools in the toolset).

A.1 Model Preprocessing Tools

Model preprocessing scripts can be used for constructing and transforming UML models before analyzing them.

A.1.1 `flatten_state_machine_hierarchy.py`

Description

Removes hierarchy from state machines and normalizes transitions of state machines in a model.

On-Line Documentation

```
usage: python flatten_state_machine_hierarchy.py [options] inputmodel.xmi output
model.xmi

Flattens the hierarchy of all state machines appearing in the model in
the file 'inputmodel.xmi'. In addition, the state machines are normalized
so that each simple state either has (i) no outgoing transitions,
(ii) only signal triggered outgoing transitions, or (iii) only
outgoing completion transitions with the additional quarantee that
at least one of them will always be enabled.

The resulting model is written in the file 'outputmodel.xmi'.


options:
  -h, --help         show this help message and exit
  --dotty-file=FILE  print the state machines in dotty format to files
                     starting with FILE [default: no dotty output]
```

Example

```
$ python flatten_state_machine_hierarchy.py models/TV.xmi TV_flat.xmi
Flattening the state machine '<<anon>>::TV::statemachine of TV'
The state machine has
    15 state vertices
     1 concurrent composite states
     4 nonconcurrent composite states
     6 simple states
     0 final states
     4 initial pseudostates
     0 choice pseudostates
    13 transitions
The flattened state machine has
    11 state vertices
     0 concurrent composite states
     1 nonconcurrent composite states
     6 simple states
     0 final states
     1 initial pseudostates
     3 choice pseudostates
    24 transitions
$
```

A.1.2 `flatten_state_machine_action_language.py`

Description

Makes the control flow of conditional statements, and `while` and `for` loops explicit in the transition structure of each state machine in a model.

On-Line Documentation

```
usage: python flatten_state_machine_action_language.py [options] inputmodel.xmi
outputmodel.xmi

Removes if, for, while, do, break, and continue constructs
from the action language statements appearing in
the state machines of the model in the file inputmodel.xmi.
Writes the resulting model in the file outputmodel.xmi.

options:
  -h, --help            show this help message and exit
  --single-statement    extra flattening: each transition will have at most
                        one statement
  --atmost-one-send-statement
                        extra flattening: each transition will have at most
                        one send statement
  --dotty-file=FILE     print the state machines in dotty format to files
                        starting with FILE [default: no dotty output]
```

Example

```
$ python flatten_state_machine_action_language.py model.xmi model_flat.xmi
$
```

A.1.3 slice_state_machines.py

Description

Simplifies state machines in a model using program slicing techniques.

On-Line Documentation

```
Usage: slice_state_machines.py [OPTIONS] FILE

Slices UML state machines in the UML model in FILE.

Options:
  -o, --outputfile=FILENAME  Specifies the name for the file in which to
                             write the sliced UML model. If not specified,
                             original name will be used with '.sliced' added
                             into it. For example, a sliced version of
                             'model.xmi' would be 'model.sliced.xmi'.
                             If the name of a model to be sliced does not
                             end with postfix '.xmi' or '.xml', a sliced
                             model will have the name of the original model
                             with a '.sliced.xmi' postfix added.

  -h, --help                 Displays this text.

  -C, --cfg                  Writes dot files describing control flow graph
                             (CFG) used in slicing.

  -I, --inputsm              Writes dot files containing the original state
                             machines represented by graphs in DOT language.

  -O, --outputsm             Writes dot files containing the sliced state
                             machines represented by graphs in DOT language.

  -D  --dependencies=OPTIONS Specifies what dependencies are drawn to the
                             dot-file representing CFG (if --cfg flag is used)
                             Possibilities are:
                             d = data dependencies
                             n = non-termination sensitive control dependencies
                             o = decisive control dependencies
                             i = interference dependencies
                             s = signal sending dependencies

                             For example -Ddno prints data dependencies,
                             non-termination sensitive control dependencies,
                             and decisive control dependencies.

                             If this flag is not given, all dependencies are
                             drawn as a default.

  -N, --nosliced             Writes no sliced UML model.

  -A  --sliceassert          Adds all assert statements to slicing criterion.


Slicing criterion is defined in the UML model given by user as an
input.  It is a set of transitions that contain all the transitions
from the original UML model that have a taggedValue named
'slicingCrit'. The actual value of taggedValue 'slicingCrit' have no
meaning.

You can also add all assert-statements to slising criterion by using
the correct command line option.

Dot files are text files describing a graph in a DOT language. Graphs
described in DOT can be viewed with a small program called 'dotty' which is
part of a Graphviz package.

For more information about the theory behind the slicer, see
http://www.tcs.hut.fi/Studies/T-79.5001/reports/ojal07sliuml.pdf

This script was written by Vesa Ojala for the SMUML project
<http://www.tcs.hut.fi/Research/Logic/SMUML.shtml>.
```

Example

```
$ python slice_state_machines.py model.xmi
2 statemachine(s)
reaching definitions calculated
data dependencies calculated
non-termination sensitive control dependencies calculated
interference dependencies calculated
decisive control dependencies calculated
control flow graph slice calculated
sliced UML model generated
sliced model wrote to file 'model.sliced.xmi'

done
$
```

A.1.4 `flatten_transition_effects.py`

Description

Replaces compound arithmetic expressions in effects of transitions of state machines in a model with sequences of assignments of simpler arithmetic expressions to temporary variables added to the model.

On-Line Documentation

```
Usage: flatten_transition_effects.py INFILE OUTFILE
Flatten action language expressions in effects of transitions of state
machines in a UML model contained in a XMI 2.0 project file INFILE, and
generate a new project file and model (with possibly new temporary
variables added to its classes) as OUTFILE.  The model contained in the
input file should first have been processed through the state machine
hierarchy and action language flatteners (if necessary).

"Flattening" an expression means translating the expression to a series
of assignments to temporary variables introduced for subexpressions of
the expression.  For example, the expression "((y + z) - w) * q" could
be flattened to a plain reference to a temporary variable "tmp_2"
initialized using the sequence of assignments
"tmp_0 = (y + z); tmp_1 = tmp_0 - w; tmp_2 = tmp_1 * q;".

This script will flatten only action language expressions of integer type;
for example, the topmost "Boolean part" of any expression is left intact
(integer subexpressions of Boolean expressions are still flattened, though).

Options:
  -h, --help                Display this help and exit.

This script was written by Heikki Tauriainen for the SMUML project
<http://www.tcs.hut.fi/Research/Logic/SMUML.shtml>.
```

Example

```
$ python flatten_transition_effects.py model.xmi model_flat.xmi
$
```

A.1.5 jumbala_to_uml.py

Description

Constructs a UML model from a collection of concurrent processes specified as simple programs in the Jumbala action language.

On-Line Documentation

```
Usage: jumbala_to_uml.py [OPTION...] INPUT-FILE OUTPUT-FILE
Construct a UML 1.4 model from one or more simple sequential Jumbala
programs representing concurrent processes communicating via shared
variables.

The following command line option is supported:
  -h, --help                 Display this help and exit.

The input for the script should consist of definitions of process types
and processes.  A process type represents a simple sequential Jumbala
program with variables (each of which can be of type "int", "boolean", or
a reference to another process type); each process type will be
represented by a unique active class in the constructed model.  A process
is an instantiation of a process type and corresponds to an object in the
constructed model.  There can be multiple instantiations of the same
process type.

In short, each process type definition should begin with a line of the
form "#proctype <PROCTYPE-ID>" (where <PROCTYPE-ID> is a valid name for
a non-primitive type in the Jumbala action language) followed by a
newline-separated list of variable declarations (in the usual form,
e.g., "int x;") and a fragment of Jumbala action language code, which may
refer to the variables declared in the process type definition, but may
not contain any further variable, class, or function declarations.  The
type of a variable can be either "int", "boolean", or the name of a
process type defined in the same input file.  The name of a variable
should be a valid identifier in the Jumbala action language.

Each process definition should begin with a line of the form
"#process <PROCESS-ID> <PROCTYPE-ID>", where <PROCESS-ID> is a valid name
for an identifier in the Jumbala action language, and <PROCTYPE-ID> is a
name of a process type defined in the same input file.  If the process
type of the process declares references to other process types, all such
references have to be initialized for the process by following the
"#process ..." line with a newline-separated list of assignments of the
form "<REF-ID> = <PROCESS-ID>;", where <REF-ID> is the name of a
reference to a process type declared in <PROCTYPE-ID>, and <PROCESS-ID>
is the name of a process instatiated using this process type in the same
input file.  Primitive type variables of processes can (but do not have
to) be initialized (with a constant value of the appropriate type) using
a similar syntax "<VARIABLE-ID> = <VALUE>;".

Formally, the input for the script should conform to a string generable
from the nonterminal <definitions> using the following grammar.
(Nonterminals in the grammar should be separated from other sequences
of non-whitespace characters by white space.  Strings in quotes should
be interpreted literally without the quotes; <EOL> represents a newline.
Comments are enclosed in /* ... */.)

<definitions> ::= <definition> <definitions> | /* empty */

<definition> ::= <proctype-definition> | <process-definition>

<proctype-definition> ::= <proctype-declaration> <var-declarations>
                          <ACTION-LANGUAGE-CODE>

<proctype-declaration> ::= "#proctype" <PROCTYPE-ID> <EOL>

<var-declarations> ::= <var-declaration> <var-declarations> | /* empty */

<var-declaration> ::= ("int" | "boolean" | <PROCTYPE-ID>) <VARIABLE-ID> ";"
                      <EOL>

<process-definition> ::= <process-declaration> <initializations>
```

```
<process-declaration> ::= "#process" <PROCESS-ID> <PROCTYPE-ID> <EOL>

<initializations> ::= <initialization> <initializations>
                    | /* empty */

<initialization> ::= <var-initialization> | <ref-initialization>

<var-initialization> ::= <VARIABLE-ID> "=" <VALUE> ";" <EOL>

<ref-initialization> ::= <REF-ID> "=" <PROCESS-ID> ";" <EOL>

Note that this simple helper script will not check <ACTION-LANGUAGE-CODE>
for any errors.  Additionally, the generated model may need to be
processed through the action language flattener included in the SMUML
toolset before it is ready to be analyzed.

This script was written by Heikki Tauriainen for the SMUML project
<http://www.tcs.hut.fi/Research/Logic/SMUML.shtml>.
```

Example

```
$ cat program.jmb
#proctype Producer

int i;
int value;
boolean waiting_for_consumption;
Producer self;
Consumer consumer;

i = 0;
self.waiting_for_consumption = false;

while (true) {
  /* "produce" a value */
  value = i;
  /* wait for consumer to act */
  self.waiting_for_consumption = true;
  while (self.waiting_for_consumption)
    ;
  /* check that the consumer now has the correct value */
  assert self.value == consumer.value;
  /* increment the value to be produced */
  self.value = consumer.value;
  i = i + 1;
}

#proctype Consumer

int value;
Producer producer;

while (true) {
  /* wait until a value is ready to be consumed */
  while (!producer.waiting_for_consumption)
    ;
  /* "consume" the value */
  value = producer.value;
  /* signal that the value has been consumed */
  producer.waiting_for_consumption = false;
}

#process producer_process Producer
self = producer_process;
consumer = consumer_process;

#process consumer_process Consumer
producer = producer_process;
$ python jumbala_to_uml.py program.jmb model.xmi
$
```

A.2 Scripts for Model Analysis and Simulation

This section contains documentation for scripts which can be used for simulating and analyzing models and counterexample traces.

A.2.1 `simulator.py`

Description

Interactive simulator for UML models.

On-Line Documentation

```
Usage: simulator.py uml14model.xmi

Interactively simulates the UML1.4 model given as the argument.
```

Example

```
$ python simulator.py model.xmi
The initial trace is:
 State 0
   Object 0 p::Process
     sm_input_queue = []
     sm_defer_queue = []
     sm_state =
       top
         init

Action 1: execute transition set on object 0 (p)
  from:    top::init
  to:      top::start
Please select an action, 0 to send a signal to an object, or 'b' to backtrack on
e step, 'q' to quit: 1
 The trace is:
State 0
  Object 0 p::Process
    sm_input_queue = []
    sm_defer_queue = []
    sm_state =
      top
        init
Action: Object 0 executed transition 'initialize'
State 1
  Object 0 p::Process
    sm_input_queue = []
    sm_defer_queue = []
    sm_state =
      top
        start

Action 1: execute transition set on object 0 (p)
  from:    top::start
  to:      top::run
  effect: assert 5 < 2;
Please select an action, 0 to send a signal to an object, or 'b' to backtrack on
e step, 'q' to quit: 1

Traceback (most recent call last):
  File "simulator.py", line 62, in ?
    while sim.interactive_step():
  File "Sim/Core.py", line 3513, in interactive_step
    o.sm_state,action_list = o.sm_state.fire(o, set, self, next_global_state)
  File "Sim/Core.py", line 1304, in fire
    raise Sim.Error.ActionLanguageError(orig, e)
Sim.Error.ActionLanguageError: An assertion violation occurred when object 0 exe
cutes transition 'Model::Process::state machine::execute code' from 'Model::Proc
ess::state machine::top::start' to 'Model::Process::state machine::top::run' wit
h effect 'assert 5 < 2;'
$
```

A.2.2 `uboco.py`

Description

Translates UML models to input files for the NuSMV model checker.

On-Line Documentation

```
usage: python uboco.py [options] infile.xmi outfile.smv

options:
  --version             show program's version number and exit
  -h, --help            show this help message and exit
  --sloppy-input        accept inconsistent UML models
  --int-bits=N          number of bits in the int type [default: 8]
  --instances=N         number of instances per class [default: 1]
  --specific-instances=CLASSNAME:N
                        number of instances of a specific class
  --queue-size=N        size of event queues [default: 2]
  --specific-queue-size=CLASSNAME:N
                        size of event queue of a specific class
  --check-deadlock      check deadlocks
  --check-implicit-consumption
                        check implicit consumption of events
  --check-assertions    check Jumbala assertion errors
  --check-runtime-errors
                        check Jumbala run-time errors
  --encode-interleaving
                        encode interleaving execution semantics
  --encode-static       encode static step semantics [default: dynamic steps]
  --old-enter           encode superlinear enter predicates [default: linear]
  --allow-queue-overdraft
                        allow exceeding queue capacity in step semantics
```

Example

Translate a model into a NuSMV input file instrumented with a linear time temporal logic property for checking the model for assertion violations.

```
$ python uboco.py --check-assertions model.xmi model.smv
$
```

A.2.3 uboco_trace_to_generic_trace.py

Description

Translates NuSMV error traces into traces in the SMUML toolset's generic counterexample format.

On-Line Documentation

```
Usage: uboco_trace_to_generic_trace.py NuSMVfile NuSMVoutputfile
```

Example

Run NuSMV on an input file generated by Uboco using the commands from the file `nusmv.scr` (included in the SMUML toolset distribution), then translate the error trace into the generic counterexample format.

```
$ NuSMV -int -load nusmv.scr model.smv
FILE ->>> nusmv.scr
*** This is NuSMV 2.4.3 zchaff (compiled on Fri Sep 21 08:42:36 UTC 2007)
*** For more information on NuSMV see <http://nusmv.irst.itc.it>
*** or email to <nusmv-users@irst.itc.it>.
*** Please report bugs to <nusmv@irst.itc.it>.
[...]
There is 1 trace currently available.
See file nusmv.out for counterexample traces.
$ python uboco_trace_to_generic_trace model.smv nusmv.out
Model model.xmi
InitialConfiguration object obj#i1#cProcess umlobject DCE:C97D1998-3086-6174-C35
B-B8BB2FBBD01A
TransitionExecution object obj#i1#cProcess transition DCE:D7AA8B65-88CE-EB95-2FC
2-80A1C2972AF0
TransitionExecution object obj#i1#cProcess transition DCE:991FA2D5-0AE8-0C46-725
2-A596E8437291
$
```

A.2.4 `suboco.py`

Description

Analyzes a model using bounded model checking and a Satisfiability Modulo Theories solver. This script generates counterexamples in the SMUML toolset's generic counterexample format.

On-Line Documentation

```
usage: python suboco.py [options] infile.xmi

options:
  --version             show program's version number and exit
  -h, --help            show this help message and exit
  --sloppy-input        accept inconsistent UML models
  --check-deadlock      check deadlocks
  --check-implicit-consumption
                        check implicit consumption of events
  --check-assertions    check Jumbala assertion errors
  --check-runtime-errors
                        check Jumbala run-time errors
  --trace-file=FILE     trace output file name [default: "infile.trace"]
  --dot-file=FILE       dot output file name [default: <none>]
  --yices=FILE          path to Yices solver binary [default: "yices"]
  --cvc3=FILE           path to CVC3 solver binary [default: <none>]
  --mathsat=FILE        path to MathSAT solver binary [default: <none>]
  --stp=FILE            path to STP solver binary [default: <none>]
  --z3=FILE             path to Z3 solver binary [default: <none>]
  --format=FORMAT       solver format: "native" or "smtlib" [default: native]
  --min-bound=N         minimum BMC problem bound [default: 0]
  --max-bound=N         maximum BMC problem bound [default: 999999]
  --int-bits=N          number of bits in the int type [default: 32]
  --queue-size=N        size of event queues [default: 2]
  --specific-queue-size=CLASSNAME:N
                        size of event queue of a specific class
  --allow-queue-overdraft
                        allow exceeding queue capacity in step semantics
  --encode-interleaving
                        encode interleaving execution semantics
  --encode-static       encode static step semantics [default: dynamic steps]
  --enum-type=ENCODING  encoding of enumerations: "enum" or "bitvector"
                        [default: "enum"]
  --enter=ENCODING      encoding for enter/exit predicates: "old", "trim", or
                        "linear" [default: "linear"]
  --qpos-type=ENCODING  queue position type: "bitvector" or "integer"
                        [default: "bitvector"]
  --queue-indexing=ENCODING
                        queue index encoding: "constant" or "parameter"
                        [default: "parameter"]
  --queue-contents=ENCODING
                        encoding of queue contents: "messages" or "tags"
                        [default: "tags"]
```

Example

Check a model for assertion violations using Suboco with Yices as the SMT solver.

```
$ python suboco.py --check-assertions --max-bound=10 --yices=bin/yices model.xmi
Bound  0, gates    97 (relevant    19)  unsat
Bound  1, gates   178 (relevant    81)  unsat
Bound  2, gates   259 (relevant   145)  sat
Trace written to model.trace
Total solver CPU time 0.000 s
$
```

A.2.5 proco2.py

Description

Analyzes a model using the Spin model checker. This script generates counterexamples in the SMUML toolset's generic counterexample format.

On-Line Documentation

```
usage: python proco2.py [options] infile.xmi

options:
  --version             show program's version number and exit
  -h, --help            show this help message and exit
  --check-assertions    check Jumbala assertion errors
  --check-deadlock      check deadlocks
  --check-implicit-consumption
                        check implicit consumption
  --check-implicit-consumption-on-class=CLASSNAME
                        check whether an instance of a class "CLASSNAME" can
                        perform implicit consumption
  --check-queue-overflows
                        check queue overlows
  --queue-size=N        size of event queues [default: 2]
  --max-search-depth=N  the maximum search depth of spin [default: 1000000]
  --use-bfs             use breadth first search in spin
  --use-collapse        use the collapse mode of spin
  --use-cex-minimization
                        use the counter-example minimization option -i with
                        spin (only relevant when --use-bfs is not used)
  --spin=FILE           the spin executable [default: spin]
  --compiler=FILE       the C compiler executable [default: cc]
  --trace-filename=FILE
                        SMUML trace output filename
  --dont-print-cex      do not print the counter-example
  --verbose             produce some verbose output
```

Example

```
$ python newproco.py --check-assertions --spin=bin/spin --compiler=cc model.xmi
The trace up to and including the current global state is:
State 0
  Object 0 p::Process
    sm_input_queue = []
    sm_defer_queue = []
    sm_state =
      top
        init
Action: Object 0 executed transition 'initialize'
State 1
  Object 0 p::Process
    sm_input_queue = []
    sm_defer_queue = []
    sm_state =
      top
        start
An assertion violation occurred when object 0 executes transition 'Model::Proces
s::state machine::execute code' from 'Model::Process::state machine::top::start'
 to 'Model::Process::state machine::top::run' with effect 'assert 5 < 2;'
Aborting!
$
```

A.2.6 simulate_generic_trace.py

Description

Simulates generic counterexample traces generated by the model checking back-end scripts uboco_trace_to_generic_trace.py, suboco.py and proco2.py.

On-Line Documentation

```
Usage: simulate_generic_trace.py trace
```

Example

```
$ python simulate_generic_trace.py counterexample.trace
Processing the trace file counterexample.trace
The trace has 2 actions
Loading model model.xmi
Getting initial global configuration correspondence...
Simulating trace...
 Object 0 moves from 'init' to 'start'
 Object 0 moves from 'start' to 'run'
The trace up to and including the current global state is:
State 0
  Object 0 p::Process
    sm_input_queue = []
    sm_defer_queue = []
    sm_state =
      top
        init
Action: Object 0 executed transition 'initialize'
State 1
  Object 0 p::Process
    sm_input_queue = []
    sm_defer_queue = []
    sm_state =
      top
        start
An assertion violation occurred when object 0 executes transition 'Model::Proces
s::state machine::execute code' from 'Model::Process::state machine::top::start'
 to 'Model::Process::state machine::top::run' with effect 'assert 5 < 2;'
Aborting!
Traceback (most recent call last):
  File "/home/htauriai/smuml/cvs/SMUML/simulate_generic_trace.py", line 461, in
?
    sim = simulate(sys.argv[1])
  File "/home/htauriai/smuml/cvs/SMUML/simulate_generic_trace.py", line 358, in
simulate
    raise RuntimeError("The trace cannot be simulated")
RuntimeError: The trace cannot be simulated
$
```

A.2.7 `analyze.py`

Description

Common interface to the model analysis tools.

On-Line Documentation

```
Usage: analyze.py [OPTION]... MODEL [BACKEND-OPTION]...

Check a UML 1.4 model contained in the project file MODEL (in XMI 2.0
format) for counterexamples.

General options:
  --configuration-file=FILENAME
                            Specify alternate configuration file
                            [default: paths.conf]
  -h, --help                Display this help and exit.
  --gui                     Run the analyzer in graphical mode.
  --quiet, --silent         Suppress all messages when running in console
                            mode.
  -v, --verbose             Show verbose output (repeating the option
                            multiple times will increase output
                            verbosity).  Effective only when running in
                            console mode.

  In console mode, the script will check the given model for
  counterexamples.  If a counterexample is found, the script will print a
  generic counterexample trace to standard output (or, alternatively, save
  the trace to a file specified using the --counterexample-trace-file
  option).  In graphical mode, the script displays a GUI for choosing model
  analysis settings; this GUI will be displayed again between model
  checking runs.

Model preprocessing options:
  --flatten-hierarchy       Run the state machine hierarchy flattener
                            before model analysis.
  --no-hierarchy-flattening Do not run the state machine hierarchy
                            flattener before model analysis [default].
  --flatten-action-language Run the action language flattener before model
                            analysis.
  --flatten-action-language-split-sends
                            Run the action language flattener before model
                            analysis; break each transition whose effect
                            contains multiple 'send' statements into a
                            sequence of transitions, each of which
                            contains at most one 'send' statement in its
                            effect.
  --flatten-action-language-single-statement
                            Run the action language flattener before model
                            analysis; break each transition whose effect
                            contains multiple simple statements into a
                            sequence of transitions, each of which
                            contains at most one simple statement in its
                            effect.
  --no-action-language-flattening
                            Do not run the action language flattener
                            before model analysis [default].
  --slice-assertions        Try to simplify the model before analysis by
                            removing action language code which cannot
                            affect any assertions in the model.
  --no-slicing              Do not apply model slicing [default].
  --flatten-effect-expressions
                            Run the state machine transition effect
                            expression flattener before model analysis.
  --no-effect-expression-flattening
                            Do not run the state machine transition effect
                            expression flattener before model analysis
                            [default].

  The model preprocessors are always run in the order (i) state machine
  hierarchy flattener, (ii) action language flattener, (iii) model
  slicer, (iv) transition effect expression flattener.
```

```
Model analysis options:
  --backend=NAME             Specify a model checking back-end to use
                             [default: uboco]. The following back-ends are
                             supported:
                               proco
                                 Use the Proco back-end with Spin as the
                                 model checker.
                               suboco[:SMT-SOLVER]
                                 Use the Suboco back-end.  SMT-SOLVER
                                 selects between the SMT solvers available
                                 for use in bounded model checking (CVC3,
                                 MathSat, STP, Yices, STP, Z3).  The
                                 default is Yices.
                               uboco[:MODE[:BOUND]]
                                 Use the Uboco back-end with NuSMV as the
                                 model checker.  MODE can be either a name
                                 of one of the satisfiability solvers
                                 supported by NuSMV (ZChaff, MiniSat, Sim),
                                 in which case the model will be analyzed
                                 using bounded model checking with the
                                 chosen SAT solver (using BOUND as the
                                 bound for BMC), or "bdd", in which case
                                 the model will be analyzed exhaustively
                                 using BDD-based model checking.  The
                                 default value for MODE is ZChaff, and
                                 the default BOUND is 30.
  --counterexample-trace-file=FILE
                             If a real counterexample is found, write a
                             counterexample trace in generic format to FILE
                             [default: "-", which refers to standard
                             output].
  --simulate-counterexample  If a real counterexample is found, simulate it
                             (writing output to standard output).
  --no-counterexample-simulation
                             Do not simulate counterexamples [default].

  Any BACKEND-OPTIONs present at the end of the command line will be passed
  to the chosen back-end (Proco, Uboco, Suboco).

Abstraction refinement options:
  --refinement               Apply automatic abstraction refinement to the
                             model's integer attributes.
  --no-refinement            Do not apply automatic abstraction refinement
                             [default].
  --integer-semantics=SEMANTICS
                             Specify the integer semantics to use for
                             generating abstract types for abstracion
                             refinement.  SEMANTICS can be one of
                             "bounded", "unbounded", or "mixed" [default:
                             bounded].
  --abstract-type-smt-solver=SOLVER
                             Specify an SMT solver to use for generating
                             operations for abstract types under bounded
                             or mixed integer semantics.  SOLVER can be
                             one of CVC3, MathSAT, STP, Yices, or Z3
                             [default: Yices].
  --expression-flattening-limit=VALUE
                             Specify an integer limit for flattening
                             definitions of inlined abstract operations
                             [default: 0, which disables expression
                             flattening altogether].  Use the value "none"
                             to enable maximal flattening.

  Abstract arithmetic operations generated using bounded integer semantics
  provide safe approximations of the corresponding operations on 32-bit
  signed integers (with wraparound on overflow) used in the Jumbala action
  language.  Alternatively, the abstract operations can be made to
  approximate operations on unbounded integers by using unbounded integer
  semantics; however, in this case only models with basic arithmetic (+, -,
  *, /), logical (&&, ||, !) and comparison operations (==, !=, <, <=,
  >, >=), and the ternary conditional operator (?:) are supported as input.
  When using mixed semantics for abstract types, definitions of basic
  arithmetic and comparison operations (and coercions between abstract
  types) will be generated by approximating the corresponding operations on
```

```
  unbounded integers, and the remaining operators will be approximations of
  bounded integer operations.

This script was written by Heikki Tauriainen for the SMUML project
<http://www.tcs.hut.fi/Research/Logic/SMUML.shtml>.
```

Example

```
$ python analyze.py --verbose --backend=uboco:ZChaff model.xmi --check-assertions
analyze.py:info:using Uboco as model checking back-end
analyze.py:info:using ZChaff as SAT solver for NuSMV
analyze.py:info:using paths configuration file ‘paths.conf’
analyze.py:info:using ‘/tmp/tmpGA3WR9’ as working directory
analyze.py:info:loading model ‘model.xmi’
analyze.py:info:analyzing model
analyze.py:info:no errors found in model (bounded model checking)
$
```

A.3 Abstraction Scripts

This section lists the tools which can be used for generating abstract types and models.

A.3.1 Abstract Type Libraries

Instead of using automatic abstraction refinement, abstract models can be generated from concrete models with the help of user-defined abstract type libraries built from one or more user-defined abstract type specification files. This section documents the specification file format.

Type Specification Files

```
The basic structure of a type specification file is defined by the
grammar below.  Nonterminals in the grammar should be separated from other
sequences of non-whitespace characters by white space.  Strings in quotes
are to be interpreted literally, and the terminal symbols (can be assumed
to) have the following meaning:
    <EOL>: end of line
    <ID>: any string usable as a name for a Python identifier
    <OPERATOR-SYMBOL>: an arithmetic operator symbol in the Jumbala
                       action language
    <TEST-SYMBOL>: a Boolean operator symbol in the Jumbala action
                   language
    <ARITY>: "1" or "2"
    <STRING>: any string containing no newlines
    <INTEGER>: any integer

<type-specification-file> ::= <type-declaration>*

<type-declaration> ::= "type" <type-name> ":" <domain-specification> <EOL>
                       <type-definition>
<type-name> ::= <ID>
<domain-specification> ::= <symbolic-value-list>
                         | <symbolic-value-list> ":" <integer-value-list>
<type-definition> ::= (<operator-declaration> | <test-declaration>
                       | <coercion-declaration>)*

<operator-declaration> ::= "operator" <ARITY> <OPERATOR-SYMBOL> <EOL>
                           <operator-defs>
<operator-defs> ::= (<symbolic-value-list> ":" <symbolic-value-list> <EOL>)*

<test-declaration> ::= "test" <ARITY> <TEST-SYMBOL> <EOL>
                       <test-defs>
<test-defs> ::= (<symbolic-value-list> ":" <boolean-value-list> <EOL>)*

<coercion-declaration> ::= "coercion" <type-name> <backend-name> <EOL>
                           <coercion-defs>
<backend-name> ::= <ID>
<coercion-defs> ::= <STRING> ":" <symbolic-value-list> <EOL> <coercion-defs>
                  | "end" <EOL>

<symbolic-value-list> ::= <ID>
                        | <ID> "," <symbolic-value-list>
<integer-value-list> ::= <INTEGER>
                       | <INTEGER> "," <integer-value-list>
<boolean-value-list> ::= <boolean-value>
                       | <boolean-value> "," <boolean-value-list>
<boolean-value> ::= "true" | "false"

In the following, we use the term "type library" to refer to all types
defined in a given set of type specification files.

In brief, a type specification file contains definitions for abstract
types, operations and tests on the types, and coercions between the types.
The part following the first '#' of every input line (if this symbol is
present) is ignored.

The definition of an abstract type begins with the keyword "type"
```

```
followed by the name of the type (the names of primitive Jumbala types
"int" and "boolean" are reserved), and a comma-separated list of symbolic
names for the values in its domain separated from the type name with ":".
This list of symbolic names can optionally be followed by another ":" and
a comma-separated list of distinct integers to be used internally as the
numeric values of the corresponding elements of the symbolic domain of the
type.  (If this list is not specified, a suitable numeric domain will be
generated automatically.  The integers in the domain are guaranteed to
correspond to unique elements of the symbolic domain of the type.)

All operator, test, or coercion definitions before the next "type"
keyword in the file will be associated with this type ("the currently
active type").

An operator (test) for elements of the currently active type is declared
using the keyword "operator" ("test") followed by the arity n of the
operator or test (1 or 2), and the operator (test) symbol, which must be
an operator symbol (whose arity matches the specified one) in the Jumbala
action language. The possible value(s) yielded by the operator (test) shall
then be enumerated in the file separately for all n-tuples of input values.

A definition of a coercion from another type to the currently active type
T1 consists of the keyword "coercion" followed by the name of another type
T2 (a primitive type such as 'int' or 'boolean', or another type defined in
the type library) and the name of a "back-end" for which to define the
coercion.  This declaration is then followed by a sequence of lines
consisting of a string S containing no newlines, a ":", and a comma-separated
list L of values from the domain of the type T1. The definition of the
coercion is terminated by a line that contains the single keyword "end".

The string S is supposed to represent a condition under which a value of
type T2 will get coerced to (one of) the values in the list L.  For the
predefined back-ends "python" and "Jumbala", S must be a valid Boolean
expression template in the back-end language, i.e., a Boolean expression
in the language (with no references to variables), where the following
constructs have special semantics:
    - If the source type (T2) for the coercion is one of the primitive
      types of the Jumbala action language (int or boolean), all occurrences
      of <> will be replaced by the actual value to be coerced from T2 to
      T1.
    - If the source type (T2) for the coercion is another user-defined
      abstract type in the type library, all occurrences of <> will be
      replaced by the internal integer representation of the value (in
      the domain of T2) to be coerced from T2 to T1.  Furthermore, for
      any element E in the symbolic domain of T2, all occurrences of <E>
      will be replaced with the internal integer representation of the
      element E in the numeric domain of T2.
Furthermore, for these back-ends, the entire coercion has the following
semantics:
    - The tests of the coercion will be evaluated (after making the above
      substitutions) in their order of specification using 'eval'.  (Thus
      every test should correspond to a valid Python expression after the
      substitutions.)
    - The result of the coercion will be the set of values associated with
      the first test which evaluates to True.
    - The result of the coercion is undefined if no test evaluates to True.
      The automatically generated code (see below) will raise a ValueError
      in this case.
For other back-ends, the string S and the coercion are not interpreted in
any way by the type library.  A user-defined back-end is thus free to
define its own semantics for the string S and the coercion.

The coercions defined between types in the type library should form a
partial order between the types such that more precise types can be coerced
only into types of (strictly) less precision, i.e., cyclic dependencies
between the precision of different types are not allowed.  The API provided
for defining coercions enforces this constraint automatically.  The API
treats the primitive types as maximally precise; using any of these
types as a target type of a coercion is not permitted.
```

Example

See the examples in Sections A.3.2 and A.3.3.

A.3.2 generate_abstract_type.py

Description

Generates abstract operations for any user-defined abstract types using a Satisfiability Modulo Theories solver.

On-Line Documentation

Example

Generate the abstract version of the "less than" comparison operation for an abstract type using the Yices Satisfiability Modulo Theories solver.

```
$ cat sign.type
type Sign: POS, ZERO, NEG
  coercion int Jumbala
    <> < 0 : NEG
    <> == 0: ZERO
    true   : POS
    end
$ python generate_abstract_type.py -I "<" -s yices:bin/yices <sign.type
type Sign : POS, ZERO, NEG : 0, 1, 2
  test 2 <
    POS, POS : true, false
    POS, ZERO : false
    POS, NEG : false
    ZERO, POS : true
    ZERO, ZERO : false
    ZERO, NEG : false
    NEG, POS : true
    NEG, ZERO : true
    NEG, NEG : true, false
  coercion int python
    <> < 0 : NEG
    <> == 0 : ZERO
    True : POS
    end
  coercion int Jumbala
    <> < 0 : NEG
    <> == 0 : ZERO
    true : POS
    end
$
```

A.3.3 `generate_interval_abstraction.py`

Description

Generates abstract operations for abstract types corresponding to finite partitions of integers.

On-Line Documentation

Example

Generate the abstract version of binary addition operation for an abstract type specified as a partition of integers into intervals.

```
$ python generate_interval_abstraction.py -I "2+" "Sign:(-inf;-1],0,[1;+inf)"
type Sign : (-inf;-1], [0;0], [1;+inf) : 0, 1, 2
  operator 2 +
    (-inf;-1], (-inf;-1] : (-inf;-1]
    (-inf;-1], [0;0] : (-inf;-1]
    (-inf;-1], [1;+inf) : (-inf;-1], [0;0], [1;+inf)
    [0;0], (-inf;-1] : (-inf;-1]
    [0;0], [0;0] : [0;0]
    [0;0], [1;+inf) : [1;+inf)
    [1;+inf), (-inf;-1] : (-inf;-1], [0;0], [1;+inf)
    [1;+inf), [0;0] : [1;+inf)
    [1;+inf), [1;+inf) : [1;+inf)
  coercion int Jumbala
    <> <= -1 : (-inf;-1]
    <> == 0 : [0;0]
    <> >= 1 : [1;+inf)
    end
  coercion int python
    <> <= -1 : (-inf;-1]
    <> == 0 : [0;0]
    <> >= 1 : [1;+inf)
    end
$
```

A.3.4 `abstractor.py`

Description

Finds abstractable attributes in a model, performs type inference using user-defined constraints on the types of attributes, and generates abstract models based on the results of type inference via a source-to-source transformation. The graphical user interface to the model abstractor is described in a separate document (see the document *SMUML Model Abstractor and Type Editor* included in the toolset documentation).

On-Line Documentation

```
Usage: abstractor.py MODE [OPTION]... FILE
Extract information about abstractable attributes, perform type inference
using given constraints, or generate an abstract model using given
constraints from a UML 1.4 model contained in a project file in XMI 2.0
format.

The first parameter to the script specifies the mode in which the script
is to operate.  The following modes are available:
  -e, --extract              Read a project file FILE containing a UML 1.4
                             model and write information about the model's
                             abstractable class and signal attributes to
                             standard output [default].
  -h, --help                 Display this help and exit.
  -i, --infer                Read a project file FILE containing a UML 1.4
                             model, perform type inference using
                             constraints from the model and standard input,
                             and write a refined set of constraints to
                             standard output.
  -r, --rewrite              Generate an abstract model from the UML 1.4
                             model in a project file FILE using type
                             constraints from the model and standard input.
  --gui                      Start a graphical user interface for
                             abstracting models and editing type
                             libraries.  This option will ignore the FILE
                             given as argument; use the menus in the
                             graphical user interface to open a project
                             file.

The following command line options are supported (console mode only):
  -d, --default-type=TYPENAME
                             (to be used in combination with -i): Specify
                             the type to be assigned to those attributes
                             whose type is not otherwise constrained by
                             the type inference process.  If not given,
                             these attributes will keep their original
                             types from the model.
  -o, --output-file=FILENAME (to be used in combination with -r): Specify
                             the name for a project file in which to write
                             the abstracted model [default:
                             'abstracted.xmi'].
  -l, --expression-flattening-limit=VALUE
                             (to be used in combination with -r): Specify
                             an integer limit for flattening definitions
                             of inlined abstract operations [default: 0,
                             which disables expression flattening].  The
                             value "none" enables maximal flattening.
  -t, --rename-transitions   (to be used in combination with -r): Rename
                             transitions in the model using the action
                             language code which was substituted for
                             their effects when rewriting the model.
  -v, --verbose              Display verbose output about the type
                             inference process.

RESTRICTIONS ON INPUT MODELS
  This script supports only input models which satisfy the following
  constraints:
    * The model should define the primitive Jumbala data types "int" and
      "boolean" as UML Primitives.
    * Attributes and associations:
        - The name of every class, signal, every class and signal
```

```
        attribute, and every navigable association end defined in the
        model should be a valid identifier in the Jumbala action
        language.  There should be no duplicate definitions of classes,
        signals, attributes, or association ends by the same name.  (The
        names of classes and signals reside in separate namespaces, as
        do the attributes and association ends defined within each class
        or a signal.)
      - The type of each attribute should be one of the primitive data
        types "int" or "boolean", or a class defined in the model.  Only
        attributes of type "int" are abstractable.
      - Each class attribute may optionally specify an initializer
        (initialValue) expression, which must be a side-effect free
        expression in the Jumbala action language.
      - Each association between classes should be a 1..1 association.
  * State machines:
      - States:
          . Every state machine in the model should have exactly one
            composite state (its "top" state); all other non-
            pseudostates in the state machine should be either simple
            states or final states.
          . All pseudostates of a state machine in the model should be
            either initial or choice pseudostates.
          . Entry and exit actions and doActivities for states are
            ignored.
      - Transitions:
          . Each transition in a state machine should either have no
            trigger, or it should be triggered by a SignalEvent
            associated with a signal defined in the model.
          . The guard of each transition in a state machine should
            either be empty, or a side-effect free Boolean expression in
            the Jumbala action language.
          . The effect of each transition in a state machine should be
            an UninterpretedAction with a (possibly empty) sequence of
            statements in the Jumbala action language as its body.  Each
            statement in this sequence should be one of the following:
              (i)   an "assert" statement with a side-effect free
                    Boolean expression;
              (ii)  an empty statement (";")
              (iii) an assignment statement, whose right-hand side is
                    either an expression of the form "new C()" for a
                    class C defined in the model, or a side-effect free
                    expression; or
              (iv)  a "send" statement referring to a signal defined in
                    the model, with side-effect free parameter and
                    target expressions.
      - Side-effect free action language expressions:
          . Side-effect free action language expressions should be
            type-consistent expressions in the Jumbala action language.
            The expressions can contain any side-effect free arithmetic
            or Boolean operators, or the ternary conditional operator.
          . Every attribute reference in a side-effect free expression
            must be either of the form "x1.x2...xn" for some n >= 1, or
            "this.x1.x2...xn" for some n >= 0.  For n >= 1, the two
            forms are semantically equivalent.  If the reference occurs
            in action language code belonging to (a state machine of)
            class C, the "this" expression refers to the instance of
            the class C itself. For all 1 <= i <= n, "xi" should be the
            name of a navigable (outgoing) end of an association (or,
            alternatively, if i == n, an attribute) defined in the class
            which is reached from class C by tracking the (possibly
            empty) sequence of associations x1->x2->...->x(i-1).

TYPE CONSTRAINTS (console mode only)
  Type constraints for the -i and -r modes of operation can be specified
  by feeding the script (via standard input) a list of names of type
  specification files (to define the library of abstract types to be used
  for type inference and abstract model construction), together with a
  list of associations between UUID's of attributes in the UML model and
  abstract types defined in the type specification files.  Use the -e mode
  of operation to obtain an initial list of constraints.

  More specifically, the input to the script (in modes -i and -r) should
  consist of zero or more lines of the form
      import <filename>
```

```
  followed by zero or more lines of the form
      <uuid> <constraint-operator> <type-name>
  where
      <filename>            is the path name of a type specification file
                            to be used for building the abstract type
                            library (if the file name begins or ends with
                            white space, enclose it in single or double
                            quotes),
      <uuid>                is the UUID of a class or signal attribute in
                            the UML model,
      <constraint-operator> is one of the operators "==", "<=", or ">="
                            (excluding the quotes) separated from <uuid>
                            by white space, and
      <type-name>           is the name of an abstract type defined in one
                            of the imported abstract type specification
                            files.

  An abstract type specification file whose name does not contain any path
  separators will be searched for, in addition to the current working
  directory, in the colon-separated list of directories given by the value
  of the SMUML_ABSTRACT_TYPE_PATH environment variable.

  The constraint operators (for an attribute with UUID 'id' and a type
  name 't', respectively) have the following semantics:
      id == t : type of attribute with UUID 'id' must be equal to 't'
      id <= t : type of attribute with UUID 'id' may not be more abstract
                than 't' (but is allowed to be any type that is at least
                as precise as 't', including the original type of the
                attribute)
      id >= t : type of attribute with UUID 'id' must be at least as
                abstract as 't'
  If no type constraint is specified for the UUID of an attribute, the
  corresponding attribute will be left unconstrained.  In this case the
  type inference procedure will be allowed to assign the attribute any
  type that is at least as abstract as its original type when searching
  for a type assignment which satisfies all constraints.  For attributes
  with no type constraints generated in the type inference process, this
  type will be either the default type for unconstrained attributes (if
  specified using -d), or the original type of the attribute.

  The part following the first '#' of every input line (if present) is
  ignored.

This script was written by Heikki Tauriainen for the SMUML project
<http://www.tcs.hut.fi/Research/Logic/SMUML.shtml>.
```

Example

List the abstractable attributes in a model.

```
$ python abstractor.py --extract model.xmi >types
$ cat types
# Class attributes
#   Model::Class::x
DCE:747789EE-C107-11DB-A38E-00508DD16D61 == int
#   Model::Class::y
DCE:74778E6C-C107-11DB-A38E-00508DD16D61 == int
#   Model::Class::z
DCE:73CE44B0-C107-11DB-A38E-00508DD16D61 == int
```

Edit the output.

```
$ $EDITOR types
[... edit the file ...]
$ cat types
# Import a type library which defines the type "Sign"
import typelibrary
# Class attributes
#   abs_simple_test::test::x
DCE:747789EE-C107-11DB-A38E-00508DD16D61 == Sign
#   abs_simple_test::test::y
DCE:74778E6C-C107-11DB-A38E-00508DD16D61 >= int
#   abs_simple_test::test::z
DCE:73CE44B0-C107-11DB-A38E-00508DD16D61 >= int
```

Apply type inference to find additional constraints on the types of the model's attributes. (The type of an attribute cannot be more precise than the type of any attribute which is used in some context for computing its value.)

```
$ python abstractor.py --infer model.xmi <types
# Abstract type specification files
import "typelibrary"
# Class attributes
#   abs_simple_test::test::x
DCE:747789EE-C107-11DB-A38E-00508DD16D61 == Sign
#   abs_simple_test::test::y
DCE:74778E6C-C107-11DB-A38E-00508DD16D61 == Sign
#   abs_simple_test::test::z
DCE:73CE44B0-C107-11DB-A38E-00508DD16D61 == int
```

Rerun type inference to generate an abstract model.

```
$ python abstractor.py --rewrite --output-file abstracted.xmi model.xmi <types
# Abstract type specification files
import "typelibrary"
# Class attributes
#   abs_simple_test::test::x
DCE:747789EE-C107-11DB-A38E-00508DD16D61 == Sign
#   abs_simple_test::test::y
DCE:74778E6C-C107-11DB-A38E-00508DD16D61 == Sign
#   abs_simple_test::test::z
DCE:73CE44B0-C107-11DB-A38E-00508DD16D61 == int
$
```

A.3.5 `canal.py`

Description

Checks whether an abstract counterexample trace corresponds to a real counterexample in the concrete model.

On-Line Documentation

```
usage: canal.py [options] TRACE_FILE CONC_MODEL

Checks whether an abstract counterexample trace corresponds to a real
counterexample in the concrete model. If it does not, relevant variables
in the trace can be calculated. Relevant variables in a given point of
trace are a set of variables are most likely the source of the spurious
counterexample. Also a hint for abstraction refinement can be given
(is given by default). A refinement hint consist of variable XMIids and of
values that should be abstracted precisely in the abstract type the variable
is. This is primarily intented to be used with interval abstractions
(used in SMUML automatic abstraction refinement procedure).

arguments:
    TRACE_FILE          Abstract counterexample trace.
    CONC_MODEL          Concrete model.

options:
  --version             show program's version number and exit
  -h, --help            show this help message and exit
  -a, --check-assert    Analyze assertion errors in counterexamples.
  -i, --check-implicit-consumption
                        Analyze implicit consumptions as errors in
                        counterexamples.
  -r, --only-relevant-variables
                        Calculates only relevant variables in the case of
                        spurious counterexample, does not suggest variables
                        for refinement.
  -f, --only-feasibility
                        Check only feasibility of a counterexample.
  -q, --quiet           Suppress all messages.
  -v, --verbose         Show verbose output. When this option is used, trace
                        simulated step by step in the abstract and in the
                        concrete model are shown with relevant variables
                        marked. (Requires the calculation of relevant
                        variables)
```

Example

```
$ python canal.py --check-assert --check-implicit-consumption counterexample.trace model.xmi
Counterexample analysator is checking for following errors:
  * runtime errors (can not be turned off)
  * assertion errors
  * implicit consumptions

Analysing feasibility of the counterexample by simulating
counterexample step by step in the concrete model:
  Object 0 moves from 'init' to 'begin'
  Object 2 moves from 'init' to 'state1'
  Object 0 moves from 'begin' to 'middle'
Implicit comsumption in Object 2 can not be executed.
Calculating relevant variables
Determining variables that are suggested to be refined
Following variables are suggested to be refined:
  DCE:F907BFC9-F43A-BEAE-F95C-5A61F278D1CD: -7, 3
  DCE:C27301A3-C65F-DD56-5018-D6BA41D12ACE: -7, 3

$
```