SMUML Model Abstractor and Type Editor Manual

Juhani Peltonen
Updated by Heikki Tauriainen

January 10, 2008

Contents

1 Introduction

This document describes the graphical user interface to the SMUML model abstraction tools.

The model abstractor can be used to examine dependencies between abstractable (integer) attributes of classes and signals of UML models, to assign these attributes abstract types chosen from abstract type libraries, to automatically check consistency of user-defined and model-induced constraints on the precision of the types, to analyze contradictions arising from the constraint analysis, and, when all contradictions have been resolved, to generate abstract models for model checking. The type editor, which is part of the graphical user interface, is used to define abstract type libraries and their operations either manually or automatically.

The rest of this document is organized as follows: Section 2 describes usage of the model abstractor and Section 3 usage of the type editor.

2 Model Abstractor

The model abstractor, shown in Figure 1, consists of the following items: a menu-bar, a tool-bar, a hierarchical view of class and signal attributes in the opened UML model (the "model hierarchy"), a tabbed container of attribute constraints, and a list of types available in the currently loaded type library.

2.1 Menu-bar

The menu-bar consists of the following top-level items: *File*, *Edit*, *Type inference* and *View*.

File→Open project... opens a dialog to select a new project file to be opened, closing the one (if any) that is currently open. A shortcut for this option is located in the tool-bar. The tool supports project files in XMI 2.0 format as produced by the Coral metamodeling tool. A project file should contain a single UML 1.4 model satisfying the following requirements:

- 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.

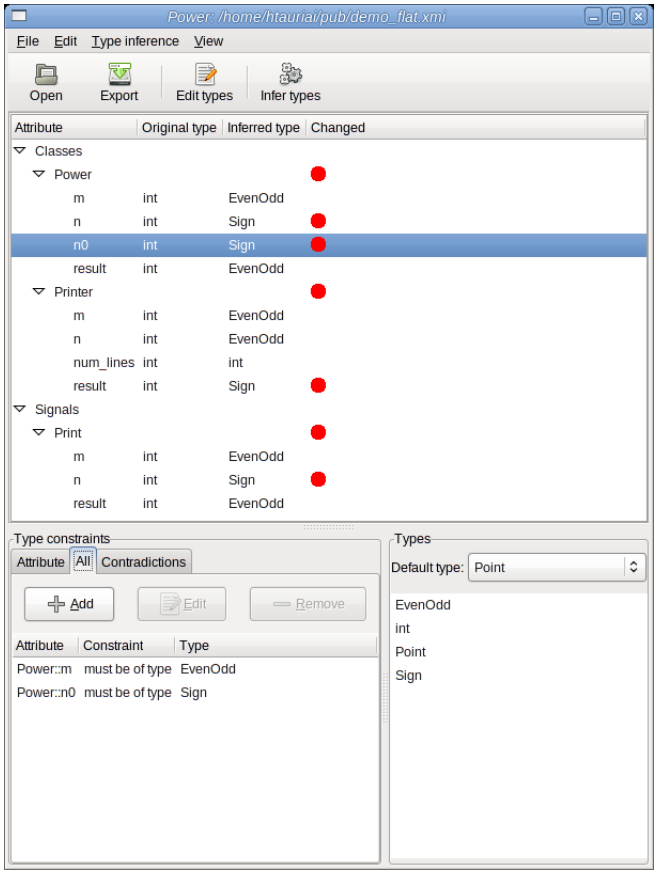

Figure 1: Main window of the model abstractor

- 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:
      - an `assert` statement with a side-effect free Boolean expression;
      - an empty statement ("`;`")
      - 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
      - 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 "$\texttt{x}_1\texttt{.x}_2\texttt{...x}_n$" for some $n \geq 1$, or "$\texttt{this.x}_1\texttt{.x}_2\texttt{...x}_n$" for some $n \geq 0$. For $n \geq 1$, 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 $1 \leq i \leq n$, $\texttt{x}_i$ 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 $\texttt{x}_1 \rightarrow \texttt{x}_2 \rightarrow \cdots \rightarrow \texttt{x}_{i-1}$.

File→Export abstract model... opens a dialog to select an existing file or to create a new one to which the currently loaded model is exported, abstracting the class and signal attributes in the model using the user-defined constraints on the types of attributes. A model cannot be exported until all contradictions in the type constraints have been resolved.

File→Close closes the loaded project.

File→Quit quits the program.

Edit→Types... opens the type editor (see Section 3). A shortcut for this option is located also in the tool-bar.

Edit→Preferences... opens a dialog which can be used to specify options which will persist between invocations of the graphical user interface. The *Options* tab of this dialog contains the following options:

Automatic type inference When checked, type inference is performed automatically after every change that is made in the attribute constraints.

Model runtime exceptions in automatic type generation When checked, results of automatically generated operations for types (see Section 3) will include the special domain value *INVALID* on those abstract operands which cover concrete operand combinations for which the corresponding concrete operation is not defined, such as division by zero. The domains of abstract types will be automatically extended with this value if necessary.

The *External tools* tab of the dialog can be used to select a Satisfiability Modulo Theories solver to use for generating definitions of operations for abstract types, and specify a path for the solver executable. This tab can also be used to specify a path to use for the `dot` tool of the GraphViz graph visualization toolset[1] used for generating images in the GUI. The *Save* button saves the preferences to an external file (`.smuml_abst_gui_prefs`) in the user's home directory. The *Apply* button applies the preferences for the loaded project but does not save them to a file. In this case the applied preferences will remain in effect only as long as the GUI is running (unless they are later changed or saved).

Type inference→Run performs type inference (a shortcut resides in the toolbar). This option is disabled if automatic type inference is active.

Type inference→Change default type... opens a dialog for selecting the default type to be used for type inference (see Section 2.5). The same task can be accomplished by choosing the default type from the dropdown menu above the list of types located in the lower right pane of the main GUI.

Type inference→Run automatically This option selects whether automatic type inference is in use or not. When checked, type inference is automatically repeated after every change that is made to the type constraints. To change the setting permanently, use Edit→Preferences.

View→Type hierarchy opens or closes a window which displays the precision ordering between integer types in the currently loaded type library. In the figure that is displayed, arrows point from more precise types to less precise ones: there is an arrow between two types if values of the source type can be directly coerced to values of the target type.

View→Attribute dependencies opens or closes a window which displays the value flow between attributes in the model (as determined from the model's action language statements and expressions). In the figure that is displayed, there is an arrow from an attribute to another one if the value of the attribute is directly used to compute the value of the other attribute in some context. When adding constraints for the types of attributes, it should be remembered that the type of an attribute can be only as precise

[1] `http://www.graphviz.org/`

as the least precise type of an attribute that is used to compute the value of the attribute in some context.

2.2 Tool-bar

The tool-bar contains shortcuts to commonly used tasks. It consists of the following items: *Open*, *Export*, *Edit types* and *Infer types*, whose actions correspond respectively to the menu items *File→Open project...*, *File→Export abstract model...*, *Edit→Types...*, and *Type inference→Run*. See Section 2.1 for a description of these options.

2.3 Model Hierarchy

The model hierarchy view contains a hierarchical view of the integer type attributes of classes and signals in the loaded model (only integer type attributes can be abstracted). The *Classes* branch contains all the classes and the *Signals* all the signals of the model. The leaves in the *Attribute* column contain the attributes. The *Original type* column contains the concrete type of an attribute which it had in the original model. The *Inferred type* shows the type of an attribute after type inference (i.e., a type suggested for the attribute to satisfy all constraints on the precision of the attributes). The *Changed* column uses a red dot to highlight those attributes whose inferred types changed in the last application of type inference (to display this information also in the case some branches in the hierarchical view are collapsed, the dot is shown also in the class and signal rows to indicate those classes and signals which contain attributes whose types changed during type inference).

2.4 Type Constraints

The *Type constraints* pane provides various views to the type constraints defined for the attributes of the loaded model. The *Attribute* tab shows the constraints associated with the currently selected attribute (if any) in the model hierarchy, the *All* tab all constraints of all attributes, and the *Contradictions* tab a set of constraints which led to a contradiction during last type inference. All tabs include an *Add*, an *Edit* and a *Remove* button. The *Add* button, which is inactive in the *Contradictions* tab, adds a new constraint on the type of an attribute. Pressing the *Add* button opens a dialog similar to the one shown in Figure 2. The *Attribute*, *Constraint* and *Type* drop-down lists identify the attribute for which the constraint is to be defined, the type of the constraint, and the constraining abstract type, respectively. The *Edit* button can be used to edit, and the *Remove* button to remove an existing constraint. Note that when removing or editing constraints to resolve type inference conflicts, the *Contradictions* tab will not be updated automatically unless automatic type inference is in use (rerun the type inference manually to check whether there remain any contradictions).

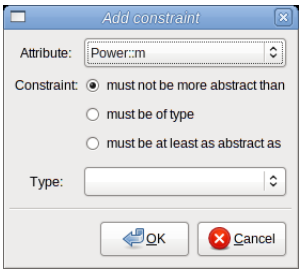

Figure 2: The "Add constraint" dialog

2.5 Types

The types view contains a drop-down list for selecting the default type for type inference. This default type will be assigned automatically to all attributes whose type is not constrained in any other way by the model-induced or user-defined type constraints. (More precisely, this means that the value of the attribute should neither depend on nor flow into any type-restricted context.) The default type can also be changed from the *Type inference→Change default type...* menu option.

3 Type Editor

The type editor, shown in Figure 3, is used to edit type libraries. The type editor can be accessed from the main GUI either by selecting *Edit→Types...* from the menu-bar or pressing the *Edit types* button in the tool-bar.

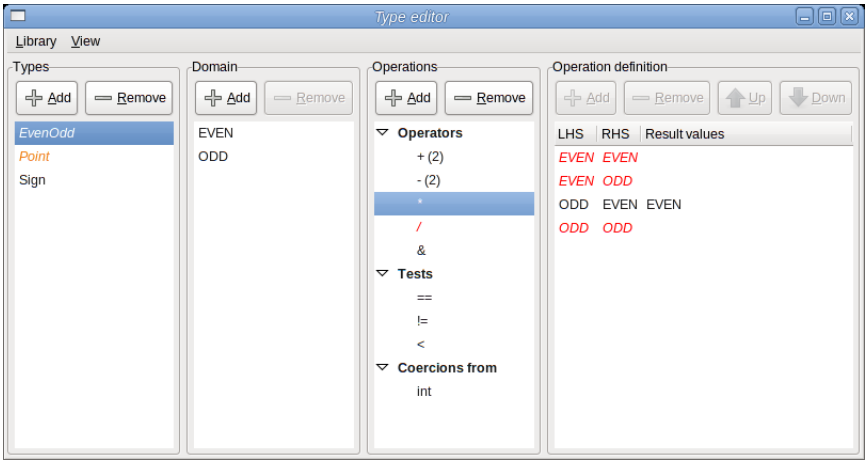

Figure 3: Type editor

The type editor consists of a menu-bar and four views labeled *Types* (which

lists all user-defined abstract integer types in the type library), *Domain* (which lists the elements in the domain of the currently selected type), *Operations* (which lists the operations defined for the currently selected type) and *Operation definition* (which shows the definition of the currently selected operation).

As seen in Figure 3, types, operations and definitions are shown in different styles according to their state. Red color on an operation indicates that the result values of the operation are undefined for all operand combinations (operand combinations with undefined result values are shown red also in the *Operation definition* view), orange that the operation is defined only partially, and black that the operation has a non-empty set of return values for all operand combinations. On a type, red color indicates that none of its operations is completely defined (in other words, the color of all operations is red), and black that all of the operations defined for the type are completely defined. Otherwise, the color of a type is orange.

All views have a context menu with varying options. This menu can be accessed via the right mouse button. The action selected from the menu applies to the item which is currently highlighted.

3.1 Menu-bar

The menu-bar consists of the following top-level items: *Library* and *View*.

Library→Import... opens a dialog to select one or more type library files to import. (To select multiple files, hold down the *Ctrl* key while selecting the files.) The imported type library will replace the currently active one.

Library→Export... opens a dialog to choose a file to which the currently active type library should be exported.

Library→Generate operations... opens a dialog to choose operations to generate automatically. This dialog is shown in Figure 4. Note that the path to the Yices executable should be set (via the *Edit→Preferences...* menu option in the main GUI window) before operations for types can be generated automatically. Furthermore, automatic generation of operations is possible only for types which have a coercion from the type "`int`" defined. (To define a coercion between user-defined abstract types automatically, both types must have such a coercion defined. Additionally, it is not possible to have cycles in the type precision hierarchy.)

In Figure 4, the "Type" drop-down list identifies the type for which to generate operations. If the box "Show only undefined operations" is checked, only those operations that are not defined for the currently selected type (shown in gray) are displayed instead of all operations. For the (partially) defined operations, a color coding similar to the one in the type editor window is used. The operations to generate can be chosen by selecting them in the list of operations.

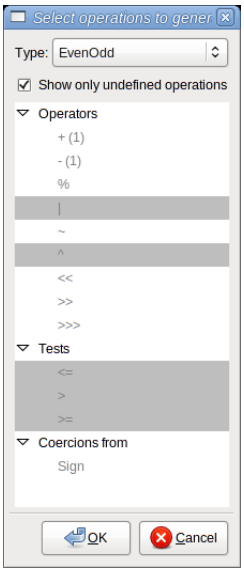

Figure 4: Operation generation dialog

If the *Model runtime exceptions in automatic type generation* option has been selected in the *Edit→Preferences...* dialog of the main GUI, automatically defined operators will include the special value *INVALID* in their set of results on those abstract operands whose concretizations cover concrete operand combinations for which the concrete operation is not defined (division or modulo by zero, bit shifts by a negative number of bits). Evaluating the concrete operation on such invalid operand combinations causes a run-time error in the Jumbala action language interpreter; the special domain value *INVALID* models the possibility of these situations explicitly. The domains of types are extended automatically with this special value both if the value is needed to represent the "result" of an operation, or if a coercion is to be defined from a type including this value in its domain.

Library→Close closes the type editor and applies the changes to the model abstractor, i.e., the *Types* view and the default type are updated and constraints referring to types that were removed from the type library are removed from the set of defined constraints.

View→Type hierarchy opens or closes the dialog which shows the precision ordering between types (described in Section 2.1).

3.2 The "Types" View

The types view shows all the user-defined abstract (integer) types in the currently active type library. A type is added by clicking the *Add* button. A sample dialog is shown in Figure 5. The name of a type should be a valid identifier in the Jumbala action language, i.e., a string conforming to the regular expression `[A-Za-z_][A-Za-z0-9_]*`.

The domain field can either be left empty or filled with one or more comma-separated values (before the the type is updated, leading and trailing white spaces are removed from the inputted domain elements). A domain element can be any string which contains no white space or the literal symbols "`,`" or "`:`". A type can be removed by first selecting it with a mouse and clicking the *Remove* button or by using the context menu (accessible using the right mouse button). The context menu also includes the *Generate operations...* option which is a shortcut to the item *Library→Generate operations...* located in the menu-bar.

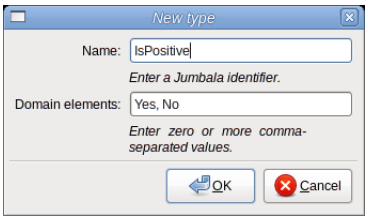

Figure 5: The "New type" dialog

3.3 The "Domain" View

The domain view shows the domain of the currently selected type. New domain elements can be added by clicking the *Add* button (see the dialog in Figure 6). As with types, several values can be added at once by separating them with commas. A domain element can be removed by selecting it with a mouse and clicking the *Remove* button or by selecting *Remove* from the right mouse button context menu when the mouse cursor points to the desired element.

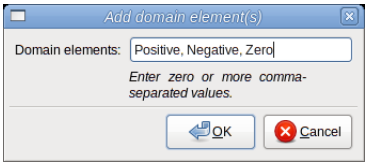

Figure 6: The "Add domain element(s)" dialog

The domain value *INVALID* has special semantics as regards automatic generation of operations for types under certain conditions (see Section 3.1).

3.4 The "Operations" View

The operations view shows the operations defined for the currently selected type. An operation can be an operator, a test or a coercion. A new operation can be added by clicking the *Add* button. A sample dialog is shown in Figure 7; operators which have variants with different number of operands are identified by showing their arities. An operation can be removed by selecting it with a mouse and clicking the *Remove* button or by selecting *Remove* from the right mouse button context menu when the mouse cursor points to the desired operation.

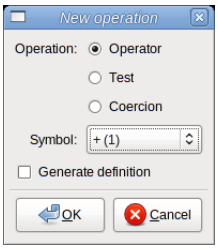

Figure 7: The "New operation" dialog

By checking the "*Generate definition*" box in the dialog, or by clicking the right mouse button on top of an operation in the list of operations shown in the type editor window and then selecting *Generate definition* from the context menu, result values are automatically filled for that operation (provided that automatic generation of operations is possible for the type, see Section 3.1).

3.5 The "Operation Definition" View

For operators and tests, the operation definition view shows all the operand combinations and their associated result values. A result value selection dialog can be opened either by double clicking a row or by selecting the *Edit...* option from the right mouse button context menu in a row whose result values should be set. The dialog is shown in Figure 8. The selection mechanism in the dialog is toggle-like: an unselected item can be selected and a selected item can be unselected by clicking it with a mouse.

For coercions, the operation definition view shows the definition of a coercion from the type currently selected in the *Operations* view (the source type) to the type currently selected in the *Types* view (the target type). The definition of a coercion consists of a list of test cases (Boolean expressions in the Jumbala action language) with associated return values. The semantics of a coercion on a value of the source type is the set of result values associated with the first test which evaluates to true when the tests are evaluated in top-down order, replacing the special construct "`<>`" in each test with the value of the source type to be coerced. (The result of the coercion is undefined if no test evaluates to true.)

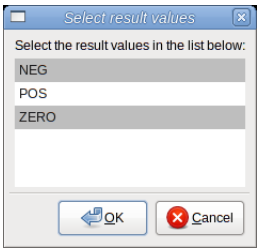

Figure 8: The "Select result values" dialog for operations

The *Add* and *Remove* buttons can be used to add and remove coercion test cases, and the *Up* and *Down* buttons (accessible also via the right mouse button context menu) can be used to change the order of the tests in the coercion.

Clicking the *Add* button (or double clicking a row in the coercion, or selecting the *Edit...* option from the right mouse button context menu) displays a dialog analogous to the one shown in Figure 9. Depending on the source type, the dialog may include an entry field for the coercion test (where "**<>**" can be used to refer to the value of the source type to be coerced), or a drop-down list of values of the source type. The result values associated with the test or value are selected from a list as for the other operations.

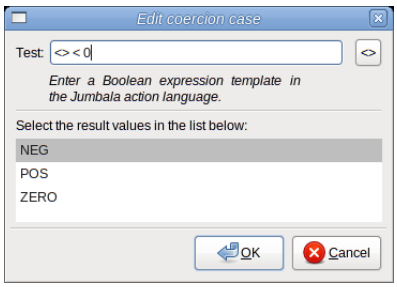

Figure 9: The "Edit coercion case" dialog