

# A short guide to the USE Model Validator

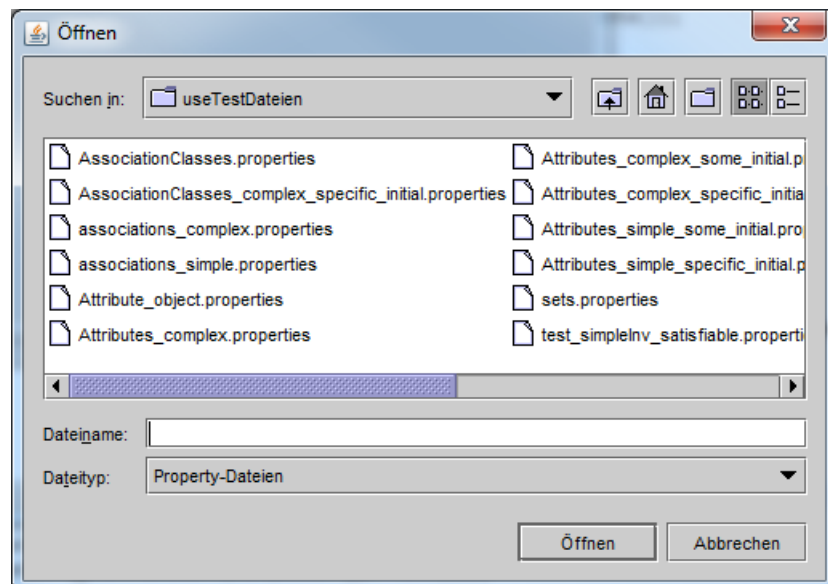
## Usage through the GUI

Note that only the most common command *validate* is available through the GUI. See section Commands for a list of all available features.

1. The plug-in can be started by either using the toolbar button in USE or by selecting the menu item “Plugins/Validation/Kodkod”.



2. If a valid USE model was previously opened, the following file open dialog is shown.



This dialog can be used to open .properties files that configure the search space. A previously constructed system state can be used as an input (as a partial solution) to the model finding process (see Commands).

3. If the model finder can find a valid solution, the corresponding system state is automatically constructed and can be displayed in a new object diagram.

## Commands

The model validator can be configured using commands on the USE shell. Using commands, the whole functionality of the model finder is available. These commands are described next.

- *modelvalidator -validate [<config-file>]*

Parameter:

- *config-file* (optional)  
Path to a configuration file (\*.properties).

Description:

If the parameter *config-file* is provided, the model validator uses the provided settings during the search process. This is identical to the usage through the GUI.

Without a configuration file the model validator uses default values for the configuration. A property file containing these values is generated using the name of the loaded USE file and the suffix *.properties*.

The used defaults are as follows:

- Exactly one object per class
- Exactly one link per association
- All attribute values must be defined
- String:  $1 \leq x \leq 5$  (Up to five different strings are used)
- Real:  $-2 \leq x \leq 2$  (Step 0.5)
- Integer:  $-10 \leq x \leq 10$
- All invariants are activated

If no solution could be found using the default values all invariants are deactivated and a second model finding run is started. The generated *.properties* file will reflect the states of the invariants.

- *modelvalidator -scrolling [<config-file>|next|previous|show(<n>)]*

Parameter:

- *config-file*  
Path to a configuration file (\*.properties).
- *next*  
Search next solution
- *previous*  
Return to previous solution
- *show(n)*  
Shows the n-th solution

Description:

The scrolling command allows searching for more than one solution. To the first call of the scrolling command a configuration file needs to be provided. Afterwards, one can scroll through the solutions by providing the argument *next*. The following solutions have at least minor differences to the previous found ones. Returning to a solution can be done by invoking the commands *previous* or *show(n)*. The command *previous* returns to the solution found before, whereas *show(n)* returns the n-th solution.

Keep in mind, that this mechanism adds the already found solutions as forbidden to the search configuration. Therefore, at some time there are no more solutions.

- *modelvalidator –scrollingAll* [<config-file>|next|previous|show(<n>)]

Parameter:

- *config-file*  
Path to a configuration file (\*.properties).
- *next*  
Search next solution
- *previous*  
Return to previous solution
- *show(n)*  
Shows the n-th solution

Description:

Like *modelvalidator –scrolling*, but all possible solutions are calculated at once and are stored in memory.

- *modelvalidator –scrollingCT* [-f <classifying term file>] [<config-file>|next|previous|show(<n>)]

Parameter:

- *config-file*  
Path to a configuration file (\*.properties).
- *next*  
Search next solution
- *previous*  
Return to previous solution
- *show(n)*  
Shows the n-th solution

Description:

The command allows to scroll through the solution space using classifying terms – a form of equivalence partitioning on UML and OCL models. The usage is like *modelvalidator –scrolling* but you are interactively asked for classifying terms (OCL queries) that you want to use. Optionally, these terms can be named for an easier identification when the tool displays the values in the current solution for them. Providing a file using the -f parameter, allows the command to run automatically without user interaction (for example in automated scripts). The file can contain names (optional) and OCL queries divided by '~~~'. An example file with two classifying terms can look like:

```
[nameOfCT1] <classifying-term-expression1>
~~~
<classifying-term-expression2>
```

- *modelvalidator –scrollingAllCT* [-f <classifying term file>] [<config-file>|next|previous|show(<n>)]

Parameter: see command *modelvalidator –scrollingCT*

Description:

Like *modelvalidator –scrollingCT*, but all possible solutions are calculated at once and are stored in memory.

- *modelvalidator -invIndep [<invName>]*

Parameter:

- *invName* (optional)  
Name of an invariant to check independence for.  
Syntax is `className::invariantName`

Description:

Checks the independence of a single invariant, if an invariant name is provided as an argument. If no name of an invariant is given, all invariants are checked step by step.

- *modelvalidator -config [satsolver := <name>; bitwidth:=<value>; automaticDiagramExtraction:=[on|off]; save]*

Parameter:

- *satsolver := name*  
Configures the SAT-solver to use (default: DefaultSAT4J).
- *bitwidth := value (1 <= value <= 32)*  
Configures the bit-width (default: 8).
- *automaticDiagramExtraction := on|off*
- *objExtraction := on|off*  
De-/Activates the extraction of a present system state as a partial solution provided to the model validator (default: off).
- *save*  
Saves the current settings.

*All arguments can be mixed and left out as required. They are separated by a semicolon (;).*

Description:

Using this command the model validator can be configured as described next.

SAT-solvers supported by Kodkod are the following (these names can be used as the value for *name* of the parameter *satsolver*):

- DefaultSAT4J
- LightSAT4J
- MiniSat
- MiniSatProver
- ZChaffMincost
- CryptoMiniSat
- Lingeling

DefaultSAT4J and LightSAT4J can be used in any environment. Currently, MiniSat und MiniSatProver are contained in the Kodkod distribution for 32-Bit environments. They can only be used, if the Java virtual machine is also running as a 32-Bit version.

The parameter `automaticDiagramExtraction` or in short `ADE` is used to configure if the model validator respects a currently present system state or if it ignores it. Using the value *on*, a previously set-up system state is used as a partial solution to the model finding process. The model validator tries to complete the solution by adding new elements.

- *modelvalidator ? [enable/disable/enabled/<ocl-expression>]*

Parameter:

- `enable`  
Enables the query mechanism
- `disable`  
Disables the query mechanism
- `enabled`  
Shows the current state of the query mechanism
- `ocl-expression`  
An OCL expression that is evaluated against the relational model used by the validator.

Description:

This command allows formulating queries against the relational solution. To do so, the query mechanism must be enabled before the validator searches for a solution. Because additional relations are required for the query functionality the runtime performance might be slower. After enabling and searching for a solution, an OCL expression provided to this command is translated into relational logic and evaluated against the relational solution. The expressions can only contain elements that were created by the validator. Manually added elements are not supported, because they are not present in the relational solution.

- *constraints -load <filename>*

Parameter:

- *filename*  
Path to a file containing additional invariants.

Description:

Loads additional invariants from the file provided by *filename*.

- *constraints -unload [<invName>]*

Parameter:

- *invName* (optional)  
Invariant name using the syntax: `classname::invariantname`

Description:

Unloads all separately loaded invariants or a single invariant if a name was provided.

## Configuration using a property file

The following settings can be made for the different elements.

### Basic types

- *String = Set{'ada'}*  
Concrete string values that are used in a solution.
- *String\_min = 1*  
Minimum number of present string values.
- *String\_max = 2*  
Maximum number of string values.
- *Integer = Set{1,2}*  
Concrete values of the type integer that need to be present in the solution.
- *Integer\_min = -10*  
Lower bound for integer values.
- *Integer\_max = 10*  
Upper bound for integer values.
- *Real = Set{15.5}*  
Concrete real values that need to be present in the solution.
- *Real\_min = -2*  
Lower bounds for real values.
- *Real\_max = 2*  
Upper bounds for real values.
- *Real\_step = 0.5*  
Step size for the interval values of real.

### Classes

- *classname = Set{ada,bob}*  
Concrete instance names for the objects of a class that need to be present in the solution.
- *classname\_min = 2*  
Minimum number of instances for the class. Concrete object names override this setting.
- *classname\_max = 4*  
Maximum number of instances for the class. Concrete object names override this setting.

### Attributes

- *classname\_attributename = Set{'Main Road', '5<sup>th</sup> Eve'}*  
Possible values for the attribute.
- *classname\_attributename\_min = 1*  
Minimum number of defined values for the attribute considering all instances of the class. The Value -1 (default) forces defined attributes for all instances of the class. This setting overrides the setting *classname\_attributename\_max* (see below).
- *classname\_attributename\_max = 3*  
Maximum number of defined values for this attribute for all instances of the class. The value -1 (default) does not constrain this number.
- *classname\_attributename\_minSize = 1*  
Minimum number of containing elements for collection based attributes.
- *classname\_attributename\_maxSize = 3*  
Maximum number of containing elements for collection based attributes. The value -1 does not constrain this number (default).

## Associations

- *associationname = Set{(ada,bob),(cyd,dan)}*  
Concrete links that need to be present in the solution. The identifiers used have to be defined as described in the classes section.
- *associationname\_min = 2*  
Minimum number of links for the association. Concrete links override this setting.
- *associationname\_max = 6*  
Maximum number of links for the association. Concrete links override this setting. The value -1 does not constraint the maximum number (default).

## Association classes

- *associationclassname\_ac = Set{b1}*  
Concrete names of instances that need to be present in the solution.
- *associationclassname\_attributename = Set{}*  
See attributes.
- *associationclassname\_attributename\_min = 1*  
See attributes.
- *associationclassname\_attributename\_max = 3*  
See attributes.
- *associationclassname = Set{(b1,bsag,hvv)}*  
Concrete links of association class that need to be present in the solution. The identifiers used have to be defined as described in the classes section.
- *associationclassname\_min = 1*  
Minimum number of link objects for this association class in the solution. Concrete links override this setting.
- *associationclassname\_max = 1*  
Maximum number of link objects for this association class. Concrete links override this setting. The value -1 does not constrain the maximum number (default).

## Invariants

- *classname\_invariantname = active/inactive/negate*  
De-/activates or negates the invariant with the given name. Setting these values in the USE invariant view takes precedence over this configuration.

## Miscellaneous

- *aggregationcyclefreeness = on/off*  
If set to on, no cycles are allowed inside of aggregations and compositions. Otherwise, cycles are allowed. (Default = on)
- *forbiddensharing = on/off*  
If off, objects participating in a composition can be member of more than one composition. Otherwise this is forbidden (Default = on).