Manual for *SU-statemodeler*

Version 1.10/1.0.12

Summary:

- graphical editor for UML-statecharts
- code-generator for Java, C++ and C#
- available as a plug-in for Eclipse
- integrated recognition of errors
- facilitates modularization and reuse
- easy to use

Contents

1 General

1.1 Overview

***S**(imple) **U**(ML)-statemodeler* is an application for the development of state-based software systems. The application consists of a graphical editor for UML-statecharts and a code-generator. With the editor every kind of state-based system can be modeled. These models can be compiled into code of several programming languages. To use *SU-statemodeler*, no programming skills are required. The software is available as standalone editor and as plug-in for Eclipse.

This manual describes how to operate *SU-statemodeler*. Essentially this includes using the editor and dealing with the generated code.

1.2 License

All *SU-statemodeler*-products (→ 5) and this manual are published under the following license:

1.3 Versions

The version numbers in this manual always apply to the standalone editor and the plug-in for Eclipse. See also chapter 5.

1.4 Contact

The author and distributor of the software is Christian Pauli. He can be contacted under christian.pauli@christianpauli.de.

2 Basics

2.1 States

A state-based software system, also called state machine, has clearly defined and finitely many different states. If a state arises then it is active and determines the behavior of the software system. A state can have subsidiary states which are called child states. According to this a state with child states is the parent state of these child states. Two states that have the same parent state are called sibling states. If a state is active then its parent state is active, too. If a state is not active then its child states are not active, too.

2.2 Pseudo states

Pseudo states are no real states, because they only influence the behavior of transitions (→ 2.3). Pseudo states are:

- decision
- join
- fork
- history state
- initial state
- end state

A **decision** influences the behavior during a transition by defining conditions that determine the new target state of the transition.

A **fork** can define more than one target state of a transition that are active at the same time. A **join** does the contrary: it determines one target state when a transition has more than one source state.

A **history state** is a the state which had been active before its parent state became inactive.

An **initial state** activates the first state of a state machine and is the origin of the behavior of the system. Every parent state can only have one initial state. There should always exist a transition originating from an initial state.

If a state machine takes its **end state**, there will be no more transitions. No transition can originate from an end state. Every parent state can have one or more end states.

While a state can be active for a certain period, a pseudo state is principally never active. From pseudo states only auto-transition can start from.

2.3 Transitions

2.3.1 Regular transitions

A (regular) transition is basically the change from one state to another. It always has a source state and a target state. Both can be the same. The source state is left and becomes inactive. The target state is entered and becomes active. An external transition is triggered by an event, additionally a condition can influence the transition: if the condition is not true, the change of the state does not take place. At last a transition can execute an action. An external transition can only be triggered if the source state is active. An external transition without an event as trigger is called **auto-transition**. An auto-transition does only happen if no child state within the source state is active, the source state itself is active and all activities started by the source state are finished. If source state and target state are the same then the state is deactivated and activated again. Transitions are used to change the behavior of the state machine.

A transition must not have an initial state as target state. Other pseudo state can be target state and source state of a transition.

2.3.2 Internal transitions

An internal transition is always triggered by an event and can be influenced by a condition. Never leading to a state change an internal transition only executes actions. It is used to trigger an action but not to change the state of the system.

There are three kinds of internal transitions:

- `entry`
- `exit`
- `do`

The event `entry` is triggered at activating, the event `exit` is triggered at deactivating a state. In addition an action can be executed. The event `do` is triggered after `entry` and before `exit` and can only be used to start activities (→ 2.4).

2.3.3 Trigger text

The trigger text consists of three parts:

1. event
2. condition
3. action

These parts appear in the following order:

```
event [condition] / action
```

All of these parts are optional. For the trigger the same rules as for state names apply. The condition is a logical expression. The action can be a function call or an expression (e. g. `g:=7`). Syntax rules for expressions are described in detail in chapter 2.5.1.

In general all parts may appear with a transition. But with specific source or target states, not all of the mentioned parts are allowed. The following enumeration shows the restrictions for certain states.

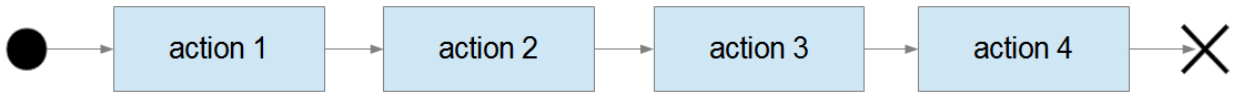
source states	**only allowed**
decision	condition
entry or exit point	action
fork	-
join	action

target states	
join	-

2.4 Activities and actions

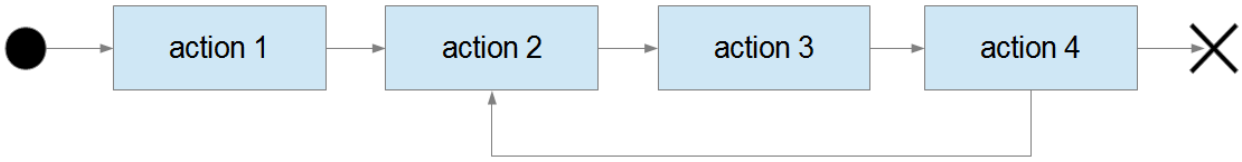
2.4.1 General

An action is an atomic, non-interruptible process. An example for an action is a function call. An action is either completely or not executed. An activity consists of several actions. It is executed by executing all its actions sequentially. An activity can be interrupted. This means that the execution is aborted before all actions have been executed. This is only possible after or before an action is executed. An exception are interruptible waiting actions.



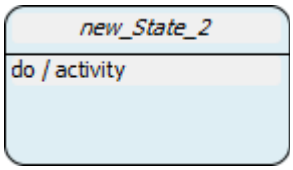
Picture 1: activity

An activity can contain cycles of actions. Action within these cycles are executed until the activity is interrupted. Thus endless activities can be created.



Picture 2: cyclic activity

Activities are used in statecharts to model parallel processes. An activity can only be executed if the state is active. If the state becomes inactive the activity is interrupted. Activities are started by internal transitions triggered by the event `do`. An activity has a name. For this name the same syntax rules apply as for state names. There can be more than one transition triggered by `do` in a state.



Picture 3: activity

2.4.2 Transition behavior

A transition from a state that has been executing activities tries to finish these activities by interrupting them. Afterward the transition must wait until all activities have actually been finished. This might take an infinite amount of time because the currently executed actions cannot be aborted immediately. Only interruptible actions can be interrupted immediately. If the implementation of an action is faulty this can lead to deadlocks.

2.5 States

A state determines the behavior of a state machine.

2.5.1 Naming rules

It must have a unique name (within the editor window) and must not be empty. A new state gets a provisional name by *SU-statemodeler*. This name can be changed by the user. The following syntax rule must apply for state names:

```
[_a-zA-Z0-9]+
```

Examples for correct names:

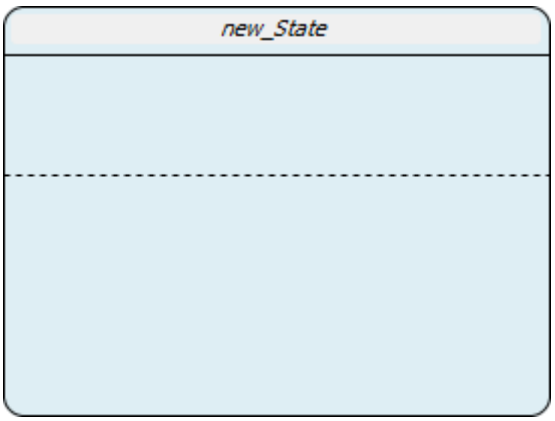
```
State_1
Device_ON
device_off
_new_State
```

Examples for incorrect names:

`state-1`	(special character)
`new state`	(blank)
`Gerät_an`	(umlaut)
`Device§on`	(special character)

2.5.2 Regions

By default only one child state is active within a parent state. If more than one child state is to be active, then the parent state must have more than one region. In every region one state can be active. Thus, complex states can be modeled. In *SU-statemodeler* regions are displayed as areas divided by dotted lines (→ Picture 4). Every region can have one initial state.

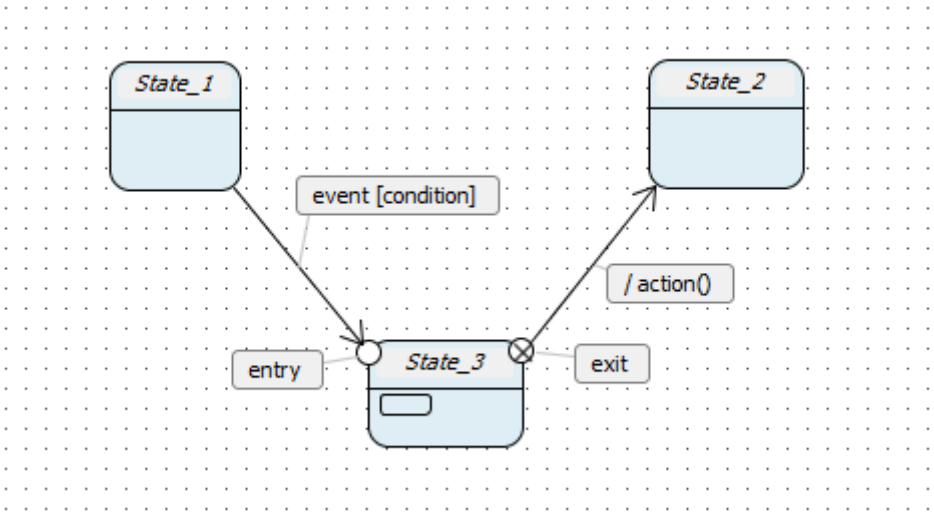


Picture 4: regions

2.6 Entry and exit points

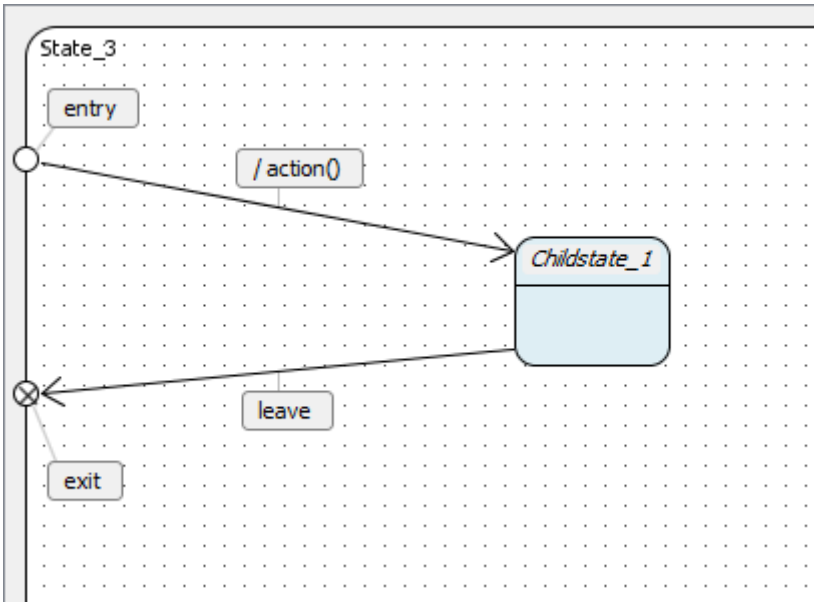
A state can have multiple entry and exit points. An entry point can be the target of a transition with a source state “outside” of the state. An entry point forwards the transition to a child state. The state itself is activated. The exit point is the counterpart of the entry point. It is the target of a transition whose source state is a child state of the state. It forwards the transition to a state “outside” of the state. The state itself is deactivated.

The following pictures show an example for an entry and exit point in *SU-statemodeler*.



Picture 5: external view `State_3`

The transition pointing to `entry` is forwarded to `Childstate1`, a child state of `State_3`. Vice versa the transition coming from `Childstate_1` pointing to `exit` is forwarded to `State_2`.



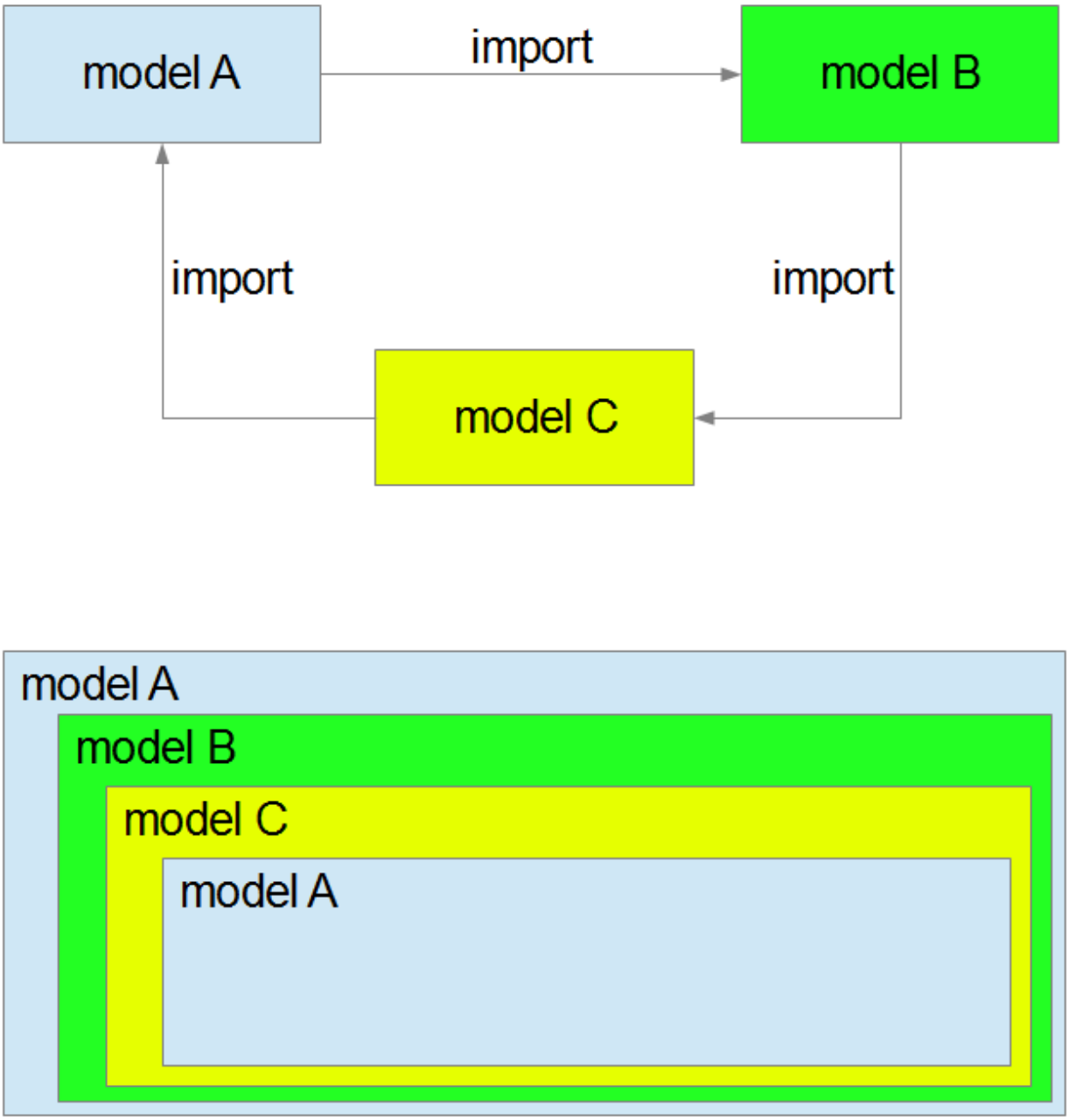
Picture 6: internal view `State_3`

2.7 Model modularization and reuse

Importing models facilitates the reuse of models and can be used to divide them into separated modules.

To import a model *SU-statemodeler* provides the import state. By this state another model (the imported model) is embedded into the current model (the importing model). The imported model becomes a sub-model of the importing model. Thus it is possible to create nested models.

When importing a model directly or indirectly into itself an import cycle occurs. Through import cycles indefinite state machines can be modeled. Picture 7 shows such an import cycle, where model A is indirectly imported into itself.



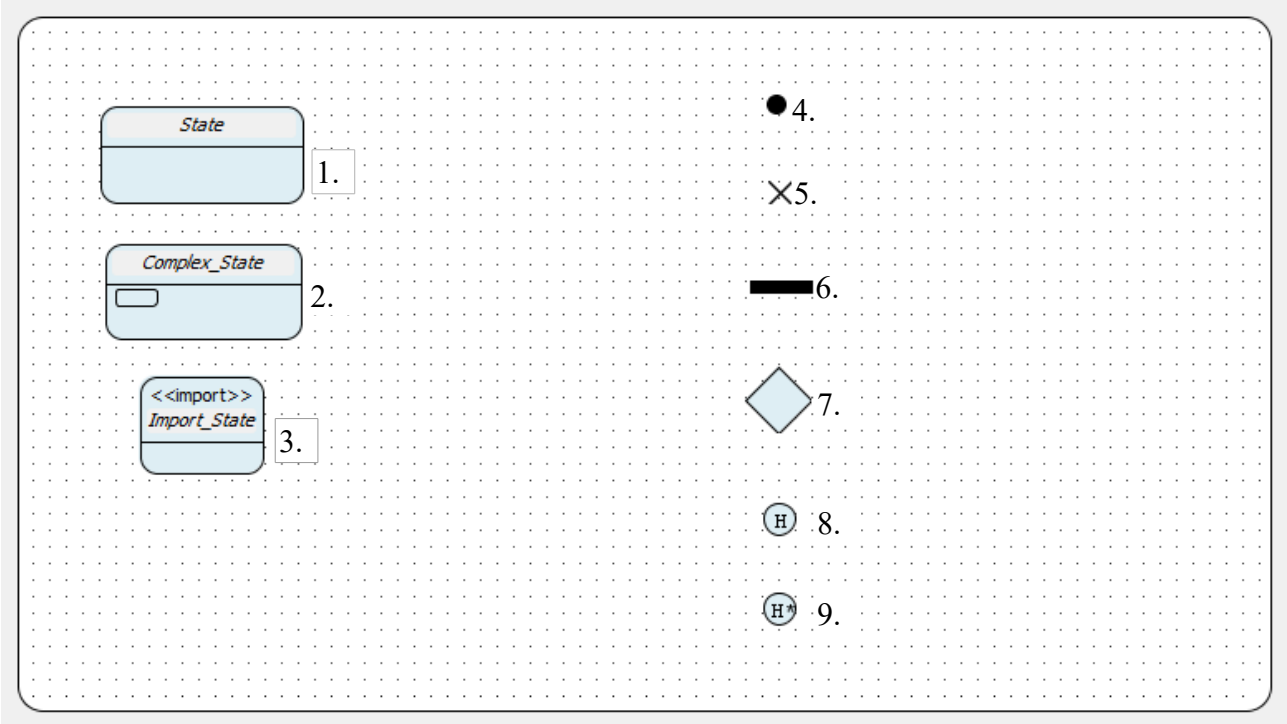
Picture 7: import cycle

3 Editor

3.1 Overview

With the graphical editor of *SU-statemodeler* state machines can be modeled with the UML-notation. The editor saves the models as XML-files (file ending *smf*). These files are called model files. The editor is used via mouse and keyboard. Particular user operations can be made undone. The editor recognizes syntax errors and certain logical errors and displays them.

Nearly everything in the editor is a state. The modeled system itself is a state with child states.



Picture 8: editor window with states

A state is displayed as a rectangle with round corners. Picture 8 shows the supported state types:

1. normal state
2. complex state
3. imported model

states

4. initial state
5. end state
6. join / fork
7. decision
8. shallow history
9. deep history

pseudo-states

A transition is displayed as an arrow, the head points to the target state. A text field shows the

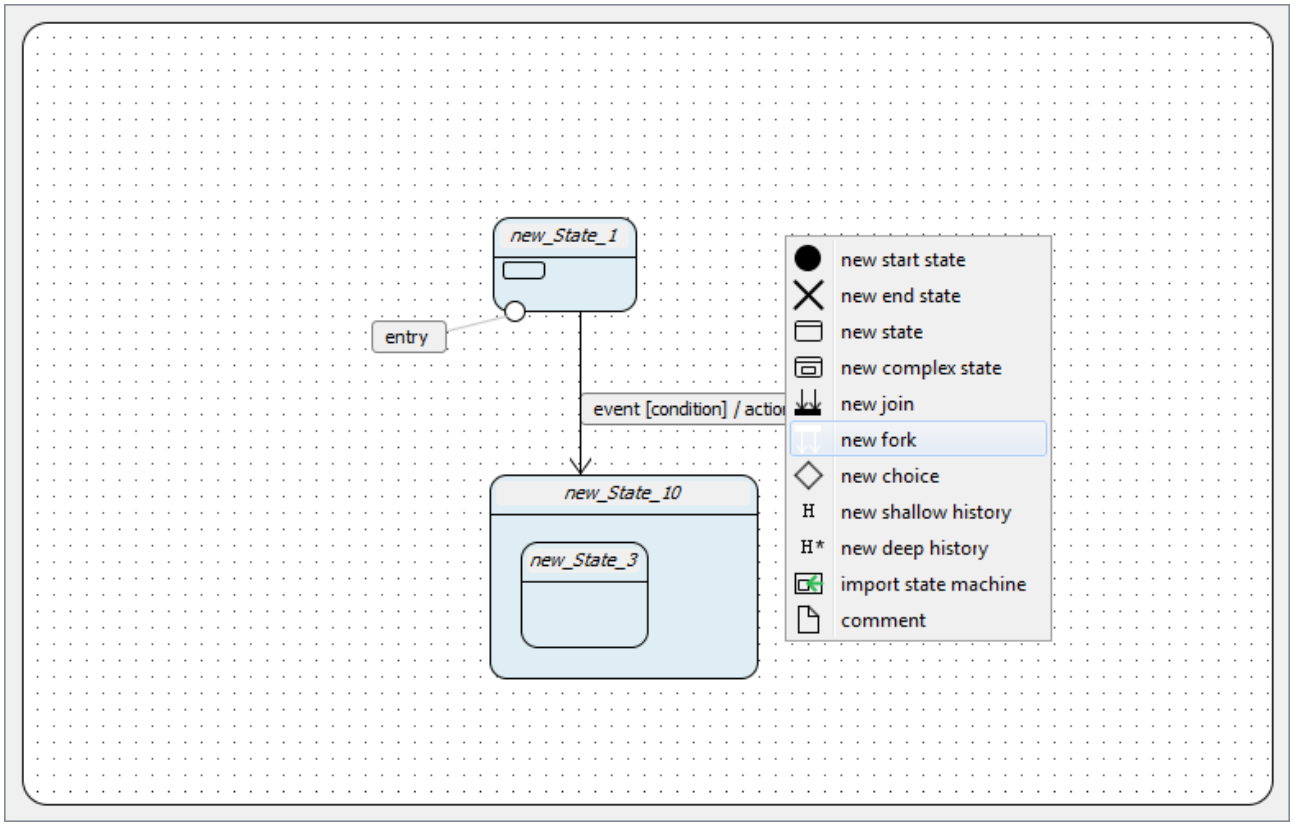
trigger text. In case of syntax errors the trigger text is colored red.

3.2 Using the editor

With the editor the user is creating UML-statecharts. In what follows all components of a statechart are denoted as *objects*.

3.2.1 Context menus

To nearly every object in the editor a context menu is available. Via this menu all relevant operations for an object (e. g. a state) can be invoked. The context menu is opened through a single right click on the object. The background has a context menu, too. Thus a context menu can be opened nearly anywhere in the editor. Picture 9 shows the context menu available at the background (indeed also a state).



Picture 9: context menu

3.2.2 Creating objects

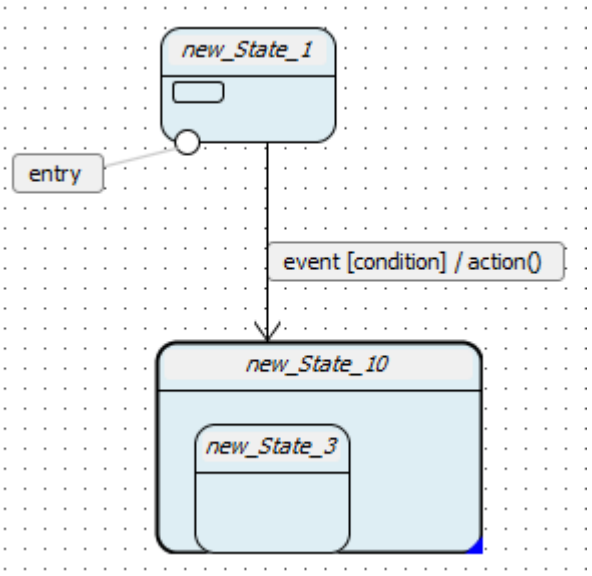
To create an object, e. g. a new state, the user has to open the context menu and invoke the according menu item (e. g. *new state*).

3.2.3 Resizing objects

Nearly every object can be resized. This does not influence the semantic of the statechart, but only the visual design.

To resize an object the user has to click the blue area normally shown in the right bottom corner of an object and drag the mouse as desired. The objects changes its size corresponding to the mouse motion unless there are restrictions concerning the size. Such a restriction are imposed by child

states: a parent state must always be large enough to include its child states (→ picture 10).



Picture 10: resizing an object

3.2.4 Moving objects

All objects can be moved. This does not influence the semantic of the statechart, but only the visual design. To move an object the user has to click and drag it (with the mouse). If a state is moved all of its child states and transitions are moved, too. A state can be move over state and region boundaries.

3.2.5 Editing text

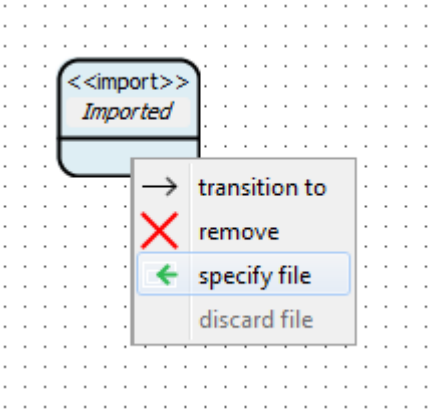
Everywhere in the editor text fields are visible, e. g. for names. To edit a text field the user has to make a double-click on it. Thus the text field is made editable and the text can be changed. Due to the size the new text needs to be fully displayed the size of the corresponding object might change.

Furthermore detached text fields can be moved like other objects.

3.2.6 Importing models

To import a model an import state is needed. An import state is created by choosing the context menu item *import state machine*.

The head of an import state is annotated with `<<import>>`, the name of the state can be changed. Then the user has to specify the model file to import. This is done via the context menu of the import state (→ picture 11). After choosing the menu item *specify file* a file dialog opens and the user can choose a model file. If this is not done the import state behaves similar to an end state.



Picture 11: import a model

The path to the file of the imported model is shown as a relative path. The file can be changed by repeating the last steps or be deleted by invoking *discard file* in the context menu.

3.2.7 Complex states

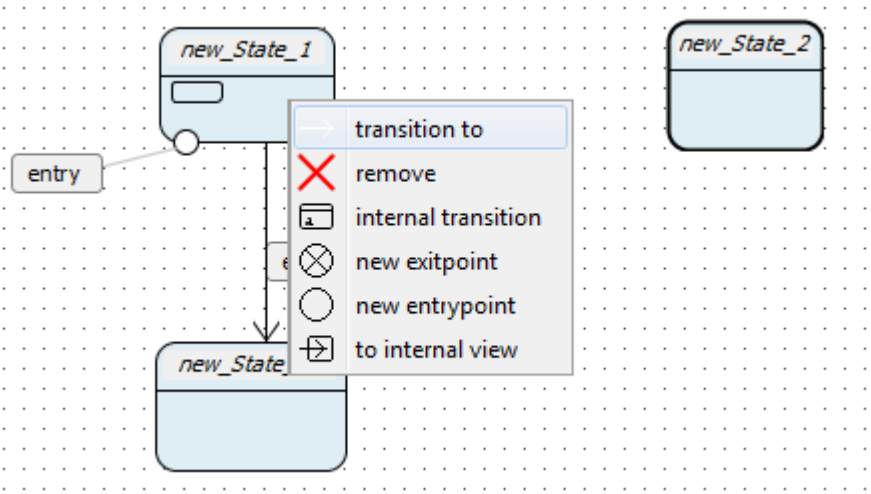
Semantically a complex state refers to a regular state. In *SU-statemodeler* a complex state is a state whose child states are not visible in the same editor window. This is for keeping the editor window as neat as possible and facilitates the creation of models with many components. The child states of a complex state are shown in a separated editor window, which is denoted as *internal view*. The window where the complex state itself is shown is the *external view*. The change between internal and external view is done via context menu or simply a double-click.

An initial state determines which child state is activated when a complex state becomes active. Furthermore entry and exit states can be attached to a complex state.

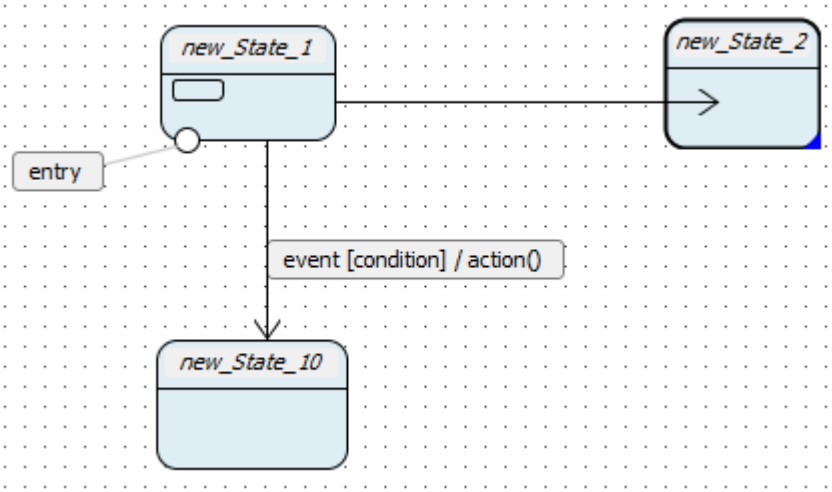
3.2.8 Transitions

3.2.8.1 Create transitions

A new transition is created via the context menu of the source state (step 1). Afterwards an arrow is displayed that moves with the mouse cursor.

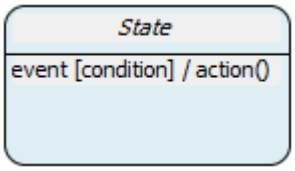


Picture 12: create new transition (step 1)



Picture 13: create new transition (step 2)

The user has to click on the target state to finish the process (step 2). Source and target state can be identical. An internal transition is also created via the context menu. The transition appears at the top of the state symbol.

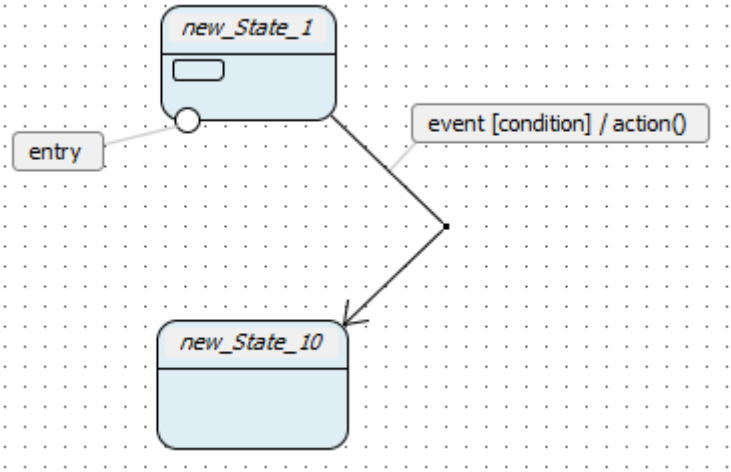


Picture 14: internal transition

Depending on source and target state the trigger text of a new transition is filled with standard values for event, condition and action.

3.2.8.2 Edit transitions

The trigger text of a transition can be changed by editing the according text field. The source and target state can be changed by dragging the according end of the transition to another object.



Picture 15: point inserted

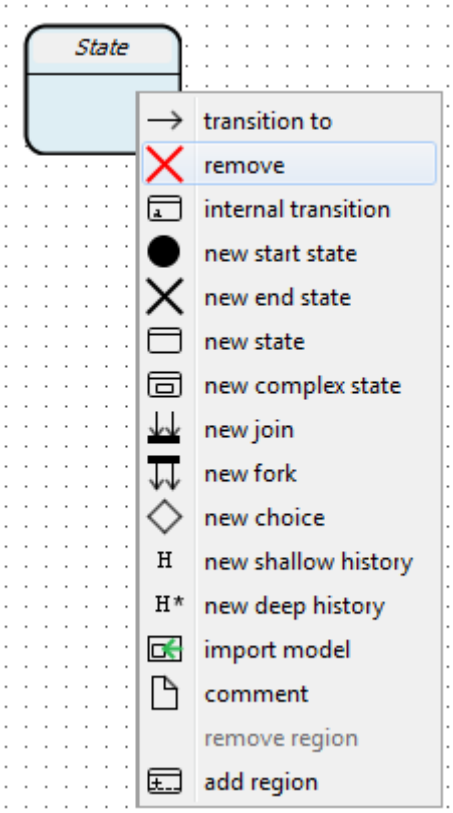
Furthermore a transition can be segmented by inserting points. This is done by a double-click on a transition. These points do not influence the semantic of a transition. They only change the visual

style of a transition, which can make a statechart more neatly.

3.2.9 Deleting objects

Every object can be deleted. If states are deleted, then its child states and all connected transitions will be deleted, too. When deleting a point of a transition (→ 3.2.8.2), then only the point itself is deleted, not the whole transition.

To delete an object the user has to open the context menu of it and choose the menu item *remove* (→ picture 16).



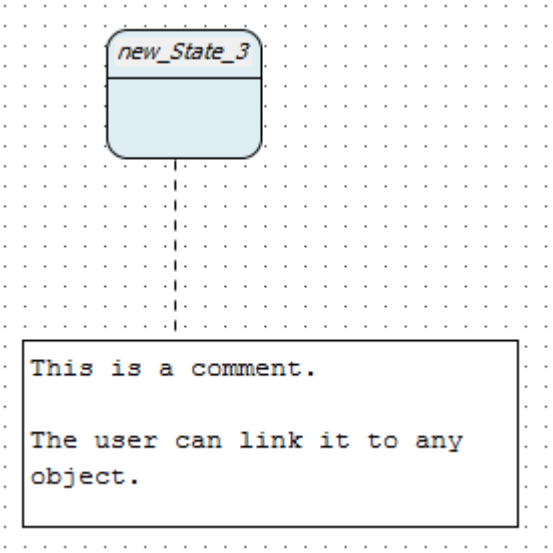
Picture 16: delete a state

3.2.10 Making changes undone

In practice unintended operation steps can happen. For this reason the editor offers the opportunity to make single operation steps undone, e. g. restore an unintentionally deleted state. A single operation step can be made undone with the keystroke combination CTRL+Z. With CTRL-Y an undone operation can be carried out again.

3.2.11 Comments

With comments the user can enrich the model with additional information. Comments do not influence semantics of the model. Comments are created via the menu item *comment* of the context menu (→ picture 16). They can be linked to any object in the editor, even to more than one.



Picture 17: comment

3.3 Expressions

3.3.1 General

Expressions are used to specify conditions and actions. A condition is a logical expression which is true or false. This can simply be the value of a variable, but also complex logical expressions consisting of operators, variables and constant values are possible.

Some examples for expressions:

`condition`	variable
`value=45`	boolean
`value="string"`	boolean
`var1=var2 & var1 > var3`	boolean
`function(34)+6=67 | value=0`	boolean
`var1:=89+var2`	assignment

3.3.2 Operators

The following operators are supported by *SU-statemodeler*:

operators		**examples**
`- +`	sign	`-8, +7.9`
`!`	negation	`!var`
`* /`	multiplication, division	`6*7`
`+ -`	addition, subtraction	`6+7, 2-1`
`= !=`	equal, not equal	`5=5, 6!=7`
`> < >= <=`	lesser, greater	`6<7, var>=0`

`&`	logical and	`a & b`
`|`	logical or	`a | b`
`:=`	assignment	`var:=7`, `var:=null`

The given order of operators is the order in which they are evaluated within an expression. Brackets can influence this order. The expression `null` represents a Null-pointer.

3.3.3 Data types

All variables and functions that are used within expressions are dynamically typed. The type of constant values, as `3` or `"string"`, is automatically determined.

The application internally distinguishes the following data types:

- short
- integer
- long

  } signed numbers

- float
- double

  } signed floating point numbers

- character — e. g. `'g'`, `'6'`
- string — e. g. `"123 abc"`
- object — abstract data types or arrays

All numbers are signed. A constant value without point is mapped to integer, a floating point number to double. Everything that is not a simple data type or a string is mapped to an object.

3.4 Error recognition

SU-statemodeler recognizes certain errors in statecharts. To each error a short description is shown in a separated panel. If a statechart contains errors, no code can be generated from it.

3.4.1 Syntax errors

If a name of trigger does not comply with the syntax rules (→ 2.5.1), the corresponding text is displayed in red color.

Picture 18: syntax error

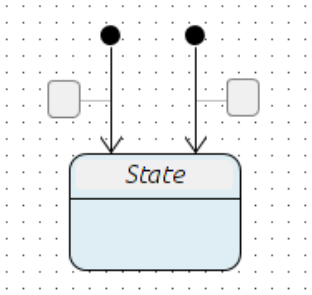
In picture 18 the name of the state is syntactically wrong, because it contains a blank character.

3.4.2 Logical errors

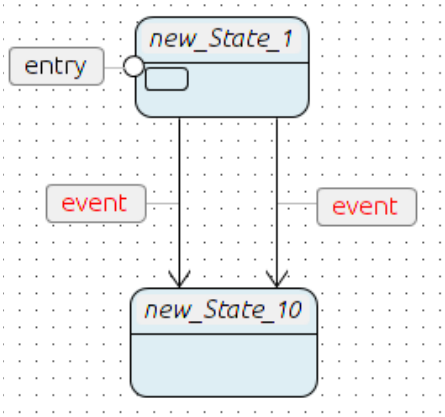
Logical errors are due to ambiguous statements or invalid target states for transitions. An example for an ambiguous statement is, when a parent state contains two initial states (within one region). Only one initial state per state and region is allowed. *SU-statemodeler* is not able to decide which initial state to use when more than one is given.

Another logical error are doubly handled events. If several transitions with the same source state are triggered by the same event, it is not clear which transition is triggered when the event occurs.

If conditions are given and if they are different, this is not an error (even if these conditions are semantically equivalent).



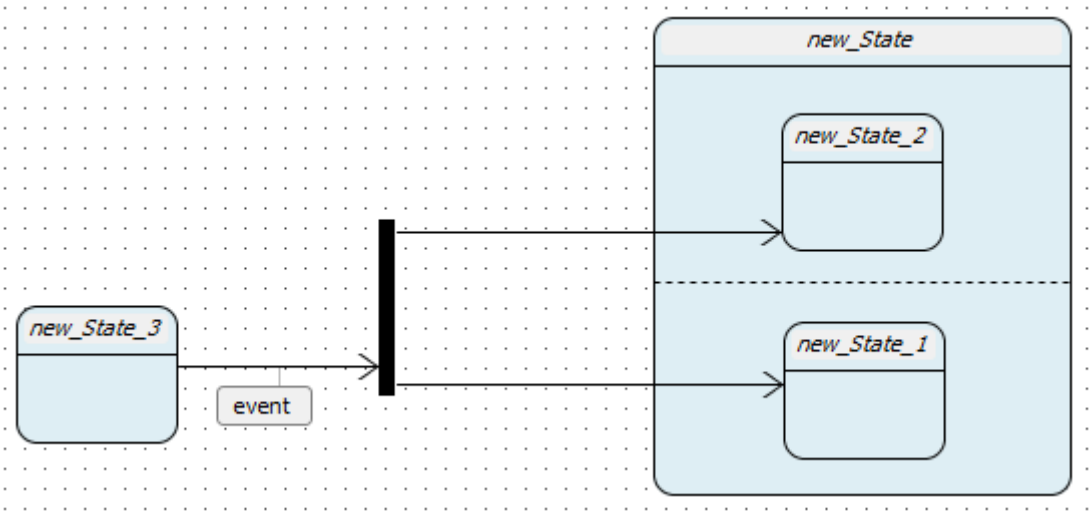
Picture 19: two initial states



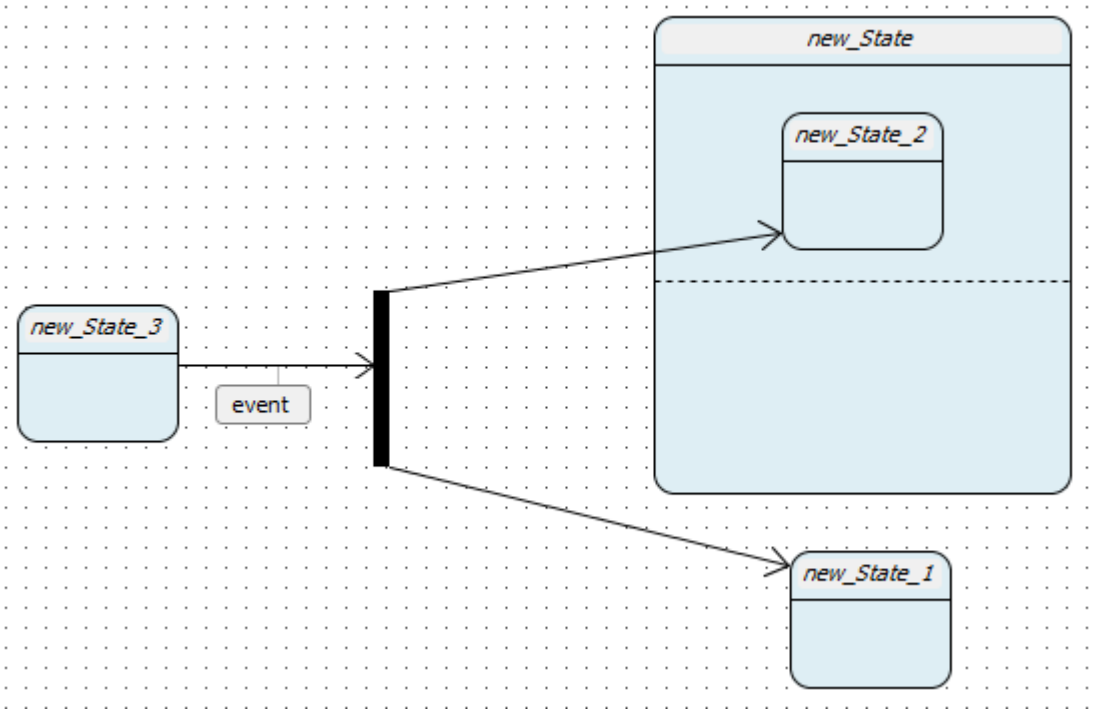
Picture 20: doubly handled event

When using forks the target states of the outgoing transitions must be in different regions of a state. This also applies to the source states of a join. A violation of these rules is recognized and shown by *SU-statemodeler*.

Picture 21 shows a correctly used fork. The following picture shows an incorrectly used fork, because one target state of the transition (`new_State_1`) is not a child state of `new_state` and hence in the wrong region.



Picture 21: fork correctly used



Picture 22: fork incorrectly used

In an editor window a state name must not occur more than once. If this rule is violated, all occurrences of the name are displayed red.

4 Code-generator

4.1 Overview

This chapter gives a short introduction about the code-generator in *SU-statemodeler*. A simple example model is introduced. In reference to that model it is explained how user-defined code can access the generated code and how to make user-defined additions to the generated code.

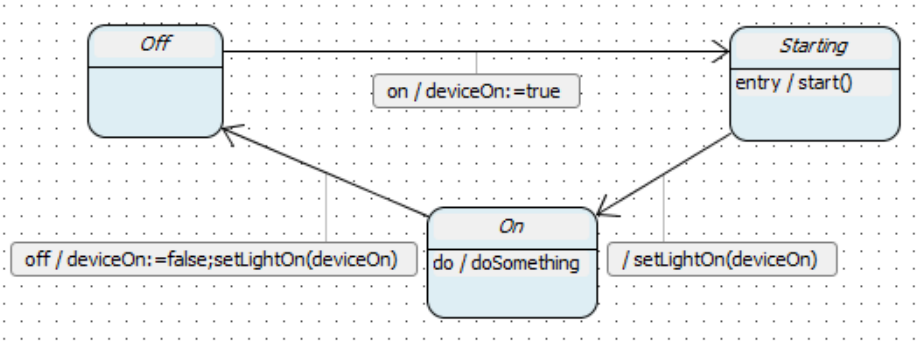
The code-generator translates the model, that was created with the editor, into executable code. By now *SU-statemodeler* can generate code for the following programming languages:

- Java
- C++
- C#

Code can only be generated if *SU-statemodeler* does not recognize any error in the model.

4.2 Example model

Picture 23 shows a simple model for a state machine having three states. For now it is not important if this system makes sense, it only is to demonstrate the general approach.



Picture 23: example model

The particular parts of the model summarized:

- 3 states `Off`, `Starting`, `On`
- 1 variable `deviceOn`
- 2 functions `setLigthOn(deviceOn: var)`, `start()`
- 1 activity `doSomething`
- 4 events `on`, `off`, `entry`, `do`

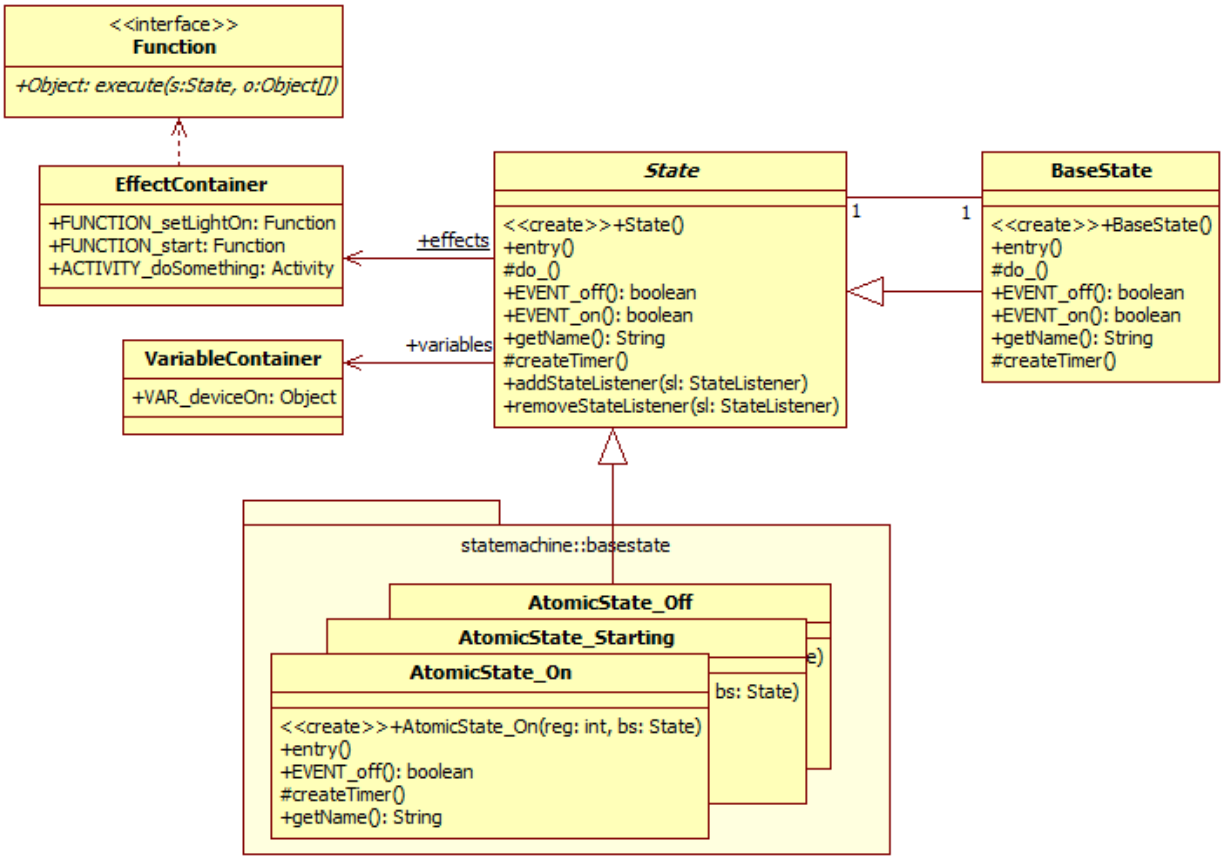
The model does not contain any syntactical error and *SU-statemodeler* will not recognize any logical error. Hence *SU-statemodeler* can generate code from it.

4.3 General characteristics

The design of the generated code follows the design pattern *STATE*[1]. For every model an abstract

1 https://en.wikipedia.org/wiki/State_pattern

class *State* is generated. This class does not implement a concrete state, but it determines the interface and the dependencies for classes implementing a state. From now on a class implementing a state is referred to as a *state*.



Picture 24: class model of generated code

Every class implementing a concrete state must be derived from *State*. This applies here for the three classes in the package *basestate*. Every class implements a state, the name of the class is derived from the state's name, led by *AtomicState_* or other prefixes. That prefix depends on the kind of state (e. g. for complex states the prefix *ComplexState_* is used). Here the class *AtomicState_On* implements the state `On`.

Furthermore the class *BaseState* is (always) generated. This class implements the state representing the whole model (respectively state machine). This means that the whole state machine is regarded as a state itself. The *BaseState* is the interface of the state machine to other components. The state machine is initialized by creating an instance of *BaseState*. Then the state machine is activated by invoking the method *entry()*. If there is no initial state, then the user has to set it. Otherwise the invocation of *entry()* has no effect. Events are triggered by invoking the according methods beginning with *EVENT_*. Example: for triggering the event `on`, user-defined code has to invoke the method *EVENT_on()*. Then the state changes to `Starting`.

A parent state holds references to its child states, one reference per region. The state implemented by *BaseState* has one region with three potential child states (in package *statemachine::basestate*). Every state has a reference to its parent state and to *BaseState*.

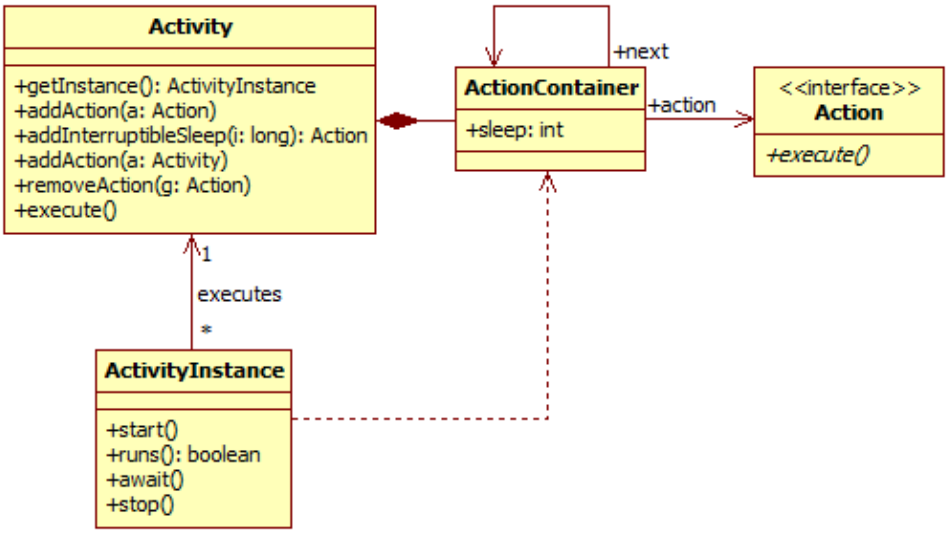
4.3.1 Variables and types

All variables used in the model (here only `deviceOn`) are stored in an instance of the class

VariableContainer. The name of the attributes corresponds to the variable names, only the prefix *VAR_* is added. All variables in *SU-statemodeler* are dynamically typed (→ 3.3.3). When compiling expressions (in the model) to code of a programming language that is statically typed (such as Java or C#), a basic type is used to implement a variable. Furthermore reflection is used to determine the type when applying operations on a variable. If the type of a variable cannot be processed by an operation, then an exception will be thrown.
Example: `z:=u+4` is a valid expression as it can be used with transitions. The operation `+` is only applicable for numbers (double, integer, short, …). The operation cannot be applied, if `u` holds a string or an object of another type.

4.3.2 Functions and Activities

Functions and activities are only declared in *SU-statemodeler*, an implementation must be provided by the user. All functions and activities are stored in an instance of *EffectContainer*. This instance is statically referenced by *State*. A function is basically a pointer to a function, which is differently implemented by the supported programming languages. A function always returns a value, which can be `null` (0). It always has the parameter *state*. It is handed over by the state machine and is the currently active state. Further parameters correspond to the parameter list of the function declaration. These parameters are represented by a static array.
An activity consists of several actions (→ 2.4). The implemented structure of activities is shown below.



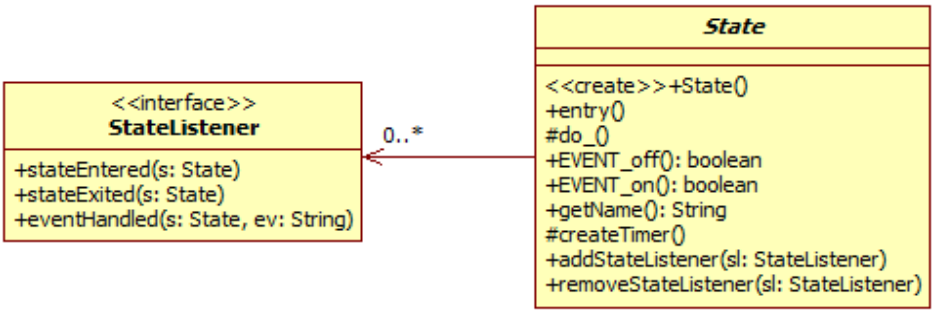
Picture 25: structure of activities

The internal structure of an activity is separated from its execution. An activity is implemented by the class *Activity*. Actions are stored in the container-class *ActionContainer*. It has the attribute `sleep`, an integer value used for waiting operations between actions. An activity can be executed more than once at the same time. The execution is implemented by the class *ActivityInstance*. It provides all functions to control the execution of an activity.
Implementing and adding actions to the activity is all that needs to be done by the user. This is done by implementing derivatives of *Action* and adding them via `addAction(a: Action)` to the activity. The code that is executed within the method `execute()` cannot be interrupted. Only wait actions can be interrupted. They wait a specified time or until they are interrupted, e. g. by an event. A wait action is added with the method `addInterruptibleAction(i: long)`. The

parameter `i` is the time to wait in milliseconds. Of course an action can wait without being interrupted, but this has to be implemented by the user.
The different supported programming languages pursue slightly different approaches for implementing actions. These approaches are described in the following chapters.

4.3.3 Listening to changes

It is possible to observe the behavior of a state machine. This is done through an observer. This observer must implement the interface *StateListener* and can be registered with the method `addStateListener(sl: StateListener)` of the class *State*.



Picture 26: state listener

The methods of the *StateListener*-interface in detail:

`stateEntered(s: State)`	Invoked, when the state `s` is activated. It is invoked after the `entry`-event was handled.
`stateExited(s: State)`	Invoked, when the state `s` is deactivated. It is invoked after the `exit`-event was handled.
`eventHandled(s: State, ev: String)`	Invoked, when an event `ev` is handled by the state `s`. It is invoked after the event was handled.

With a *StateListener* it is possible to influence the behavior of the state machine without changing the corresponding model. Other purposes can be debugging or logging.

4.4 Java

4.4.1 General

The generated code complies with Java 1.6 and later. All variables are represented by the type *java.lang.Object*. A function implements the interface *Function*. This interface only declares the method `public Object execute(State s, Object... vars)`. The parameter `vars` corresponds to the parameter list of the function declaration. If no parameter is declared then this array is `null`.

4.4.2 Implementing functions

To implement a function the user has to initialize the corresponding variables in the static reference *effects* of the class *State* with an implementation of the interface *Function*. Listing 1 shows an

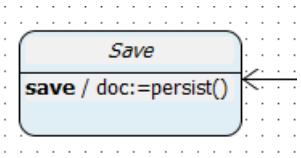
example for an implementation using an anonymous class.

```
State.effects.FUNCTION_setLightOn=new Function(){
	@Override
	public Object execute(State s,Object... vars){
		//do something here
		return null;
	}
};
```

Listing 1: function in Java

4.4.3 Persisting states

The Java code-generator comprises the function `persist` for persisting states into an XML-document. An (optional) string-parameter refers to the name of the state machine. This value is assigned to an attribute *name* of the root element. Otherwise the value is empty.



Picture 27: function persist

The function returns an instance of `org.w3c.dom.Document` or an exception[2] in case of error. It never returns `null`.

The function may be replaced by a user-defined function (→ 4.4.2).

4.4.4 Implementing actions

To implement an action the user has to define a class implementing the interface *Action*. Then an instance of it is added to the activity. Listing 2 shows an example.

```
Action a=new Action(){
	@Override
	public void execute(){
		//do something here
	}
};
State.effects.ACTIVITY_doSomething.addAction(a);
```

Listing 2: action in Java

4.5 C++

4.5.1 General

The generated code complies with C++98 and later. All variables are represented as pointers to an instance of the class *Object*. This class is defined in the header file *Object.h*, a generated file. A simple **runtime library** (RTL) manages all instances of *Object* stored in variables of the generated

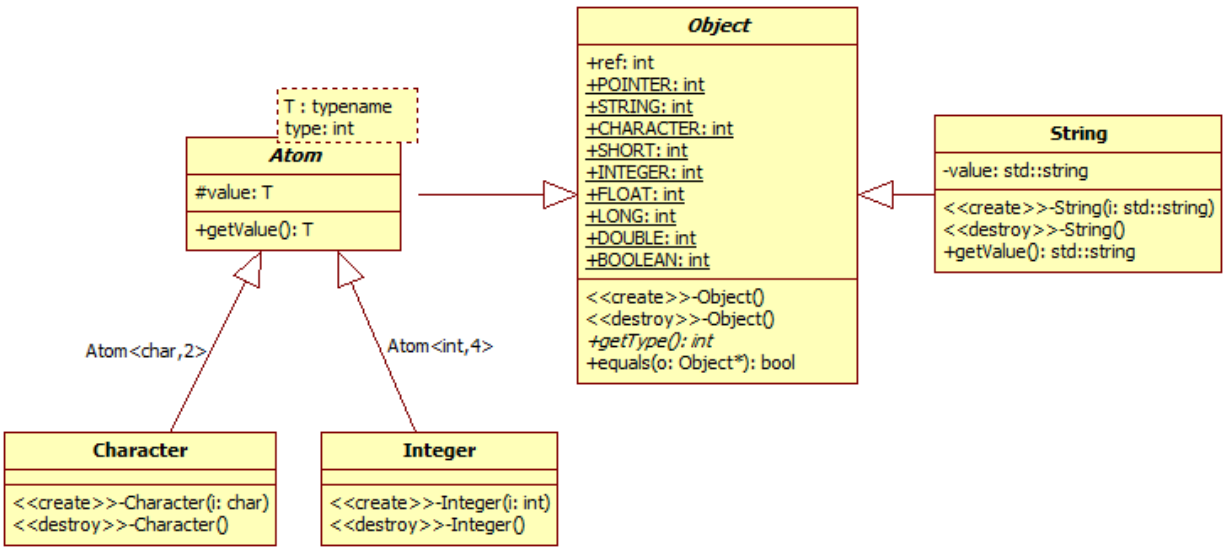
2 `javax.xml.parsers.ParserConfigurationException`

code. If an instance is not longer referenced by a variable, it is deleted from memory. This is achieved by reference counting (member `ref` in *Object*). Attention: if the user wants to keep a variable that is managed by the RTL, he needs to reference it from a variable and increment the reference counter (at least) once. Be careful with the reference counter: when it is incorrectly set, a memory leak can be the consequence.

4.5.2 Type system

The class *Object* is the base type for all supported types. An instance of *Object* cannot be created because the constructor is pure virtual. Every type has a positive integer value as unique identifier. The type of a variable can be queried at runtime by invoking the method `getType()`.

The class *Atom* represents all simple types such as `int` or `void*` by wrapping them. With `getValue()` the typed value of the object is returned. *Atom* is a template class, the template parameters are the name of the simple type and the type identifier. For all supported types the identifiers are already defined as constant class members in *Object*. In picture 28 the supported data types are shown. The class *String* wraps an instance of `std:string` from the C++ standard library.



Picture 28: type system C++

The user can extend the type system by deriving new classes from *Object* or *Atom*. To create a simple wrapper for pointers to int (`int*`) the following needs to be done:

```
//define a type derived from Object
class MyType :public Object{
private:
	int* i;

public:
	MyType(int* u){
		i = u;
	}

	int getType(){
		return 10;//new type identifier
	}

	bool equals(Object* o){
		if(o!=NULL && o->getType() == getType()){
			MyType* other = static_cast<MyType*>(o);
			return other->i == this->i;
		}
		return false;
	}
};
```

Listing 3: type implementation in C++ (variant 1)

The constructor and the method `getType()` must be implemented. The method equals need not but should be implemented. Here a more sophisticated comparison is implemented by overriding the `equals`-method.

There is another way to implement this type:

```
//define a type derived from Atom
class MyType :public Atom<int*,10>{
public:
	MyType(int* i):Atom(i){
	}
};
```

Listing 4: type implementation in C++ (variant 2)

This implementation is semantically identical to the one in listing 3, but much shorter. *Atom* already implements the desired comparison behavior, only the constructor must be implemented.

The first solution should be used with more complex types, e. g. when there is more than one attribute member. For simple types the second solution is the more appropriate one.

4.5.3 Object comparisons

Two objects can be compared with the method `equals(Object* o)` in the class *Object*. The implementation is very simple: the method only returns true when the given object is identical to the current object (i. e. `o==this`). This is not always the intended behavior so that the user might override this method in derived types. Also *Atom* only implements a very basic comparison

algorithm with the `equals`-method: the type and the value must be identical to make two objects equal.

More sophisticated comparisons for number values (double, float, short, …) are implemented by methods of the class *Operations*. By these methods even values as `5` and `5.0` are regarded as equal, while the `equals`-method of *Atom* or *Object* do not.

4.5.4 Implementing functions

All functions a derivatives of the class *Function* implementing the method `Object* execute(State* s, int lenght, ...)`. The first parameter is the current state, the second the number of the following parameters, which are actually of the type `Object*`. The header *stdarg.h* must be available, because it contains functions to obtain the parameters. Listing 5 shows how to do this.

```
//define a function class derived from Function
class MyFunction : public Function{
	public:
		Object* execute(State* s,int lenght,...){
			va_list vars;
			va_start(vars, lenght);
			Object* parameter;
			for (int a = 0; a < lenght; a++){
				parameter=va_arg(vars, Object*);
				//type checks
				if(parameter->getType() == Object::STRING){
					//cast to type
					String* str = static_cast<String*>(parameter);
					//invocation of delete is not recommended
					delete str;
				}else if(parameter->getType() == Object::INTEGER){
					//do something
					Operations::deleteObject(parameter);
				}
			}
			va_end(vars);
			return NULL;
		}
};

[...]

//initialize and assign the function
State::effects->FUNCTION_setLightOn = new MyFunction();
```

Listing 5: function in C++

To obtain the parameters the parameter list has to be initialized with the function `va_start`. Afterwards the parameters can be obtained via `va_arg(va_list list, type)`. These and other functions are declared in the header *stdarg.h*. A more detailed description for that can be found under http://www.cplusplus.com/reference/cstdarg.

Given parameters might be `null`. A check for that misses in listing 5, but should be done. Unless the values of parameters are not used somewhere else in the code, they should be freed before the

function returns. Otherwise a memory leak might occur. The internal operations of *SU-statemodeler* as (+, - or /) do that automatically. To free instances of *Object*, the method `Operations::deleteObject(Object*)` is recommended. It frees the object unless its reference counter is not zero. The invocation of the operator `delete` is possible, but not recommended. If instances of *Object* are used by user-defined code, the reference counter should be incremented once. Otherwise internal operations of *SU-statemodeler* will free them.

4.5.5 Implementing actions

The implementation of an action in C++ is derived from the class *Action*. Its pure virtual method `execute()` must be implemented. Then an instance of the class *MyAction* is added to the activity.

```
class MyAction:public Action{
public:
	void execute(){
		//do something here
	}
};

[...]

State::effects->ACTIVITY_doSomething->addAction(new MyAction());
```

Listing 6: action in C++

4.5.6 Compiling the code

The generated code can be compiled on Linux and Windows, for 32 and 64 bit. Given the appropriate libraries and compilers also other platforms might be possible. Cross-compiling is not regarded here.

4.5.6.1 Compiling on Windows

The compilation of the generated code was successfully tested on Windows 7 with the g++-compiler (version 4.8.1) and MS Visual Studio 2008. Pre-processing must be activated. When compiling the code for Windows for 64 bit, the macro `WIN64` or `_WIN64` must be defined. For 32 bit, the macro `WIN32` or `_WIN32` must be defined. Usually no additional libraries need to be configured. The header *windows.h* must be available. Using other compilers might be possible, but this was not tested.

4.5.6.2 Compiling on Linux

The compilation of the generated code was successfully tested on Ubuntu 12.04 (32 bit) with the g++-compiler (version 4.6.3). Pre-processing must be activated. The macros mentioned in chapter 4.5.6.1 must not be defined, otherwise the compilation will (probably) fail. The code uses POSIX-threads[3] as threading implementation. For this reason the header *pthread.h* must be available. The compiler also needs to know the corresponding library. That can be done with the parameter *-lpthread* (or *-pthread*) when invoking the compiler. Using other compilers might be possible, but this was not tested.

3 see http://man7.org/linux/man-pages/man7/pthreads.7.html

4.5.7 Code variants

SU-statemodeler can generate C++-code in two variants. Both variants comply with the approach as mentioned in 4.3 and are very similar.

Background: There might be problems when compiling various source files with the Visual Studio C++-compiler from Microsoft. When there are two source files having the same name, this is not a problem as long as they are in different directories. *SU-statemodeler* ensures exactly that when there are two states with the same name in a model (which leads to according source files). The Visual Studio C++-compiler from Microsoft cannot cope with this situation. The result is a linker error (LNK4042). A possible solution is to change the directory for obj-files generated by the compiler. Another solution is to tell *SU-statemodeler* to give all source files a unique name. This is done by a variant.

The first variant is referred to as *GNU*, because it is designed for the g++-compiler. Different source files can have the same name (in different directories). The second variant of the C++ code-generator is referred to as *MS*, because it is designed for the Visual Studio C++-compiler from Microsoft. Every source file has a unique name. This is achieved by appending an id to the name of the source file (e. g. *Atomic_State_Off_id2.cpp*). Of course this code can also be compiled with the g++-compiler.

4.6 C#

4.6.1 General

The generated code complies with .NET 2.0 and later. Compilation was successfully tested with Mono (version 2.8.6) and MS Visual Studio 2008. All variables are represented by the type *System.Object*. A function is represented by the delegate `public delegate object Function(State s, params object[] vars)`, an interface is not used. The parameter `vars` corresponds to the parameter list of the function declaration. If no parameter is declared then this array is `null`.

4.6.2 Implementing functions

To implement a function the user has to initialize the corresponding variables in the static reference *effects* of the class *State* with an implementation for the delegate *Function*. The following example uses a lambda expression for that.

```
State.effects.FUNCTION_setLightOn=(State st,object[] vars)=>{
      //do something here
      return null;
};
```

Listing 7: function in C#

4.6.3 Implementing actions

An action in C# is a simple delegate. It can be implemented with a lambda expression as shown in the following listing.

```
State.effects.ACTIVITY_doSomething.addAction(()=>{
      //do something here
});
```

Listing 8: action in C#

5 Products

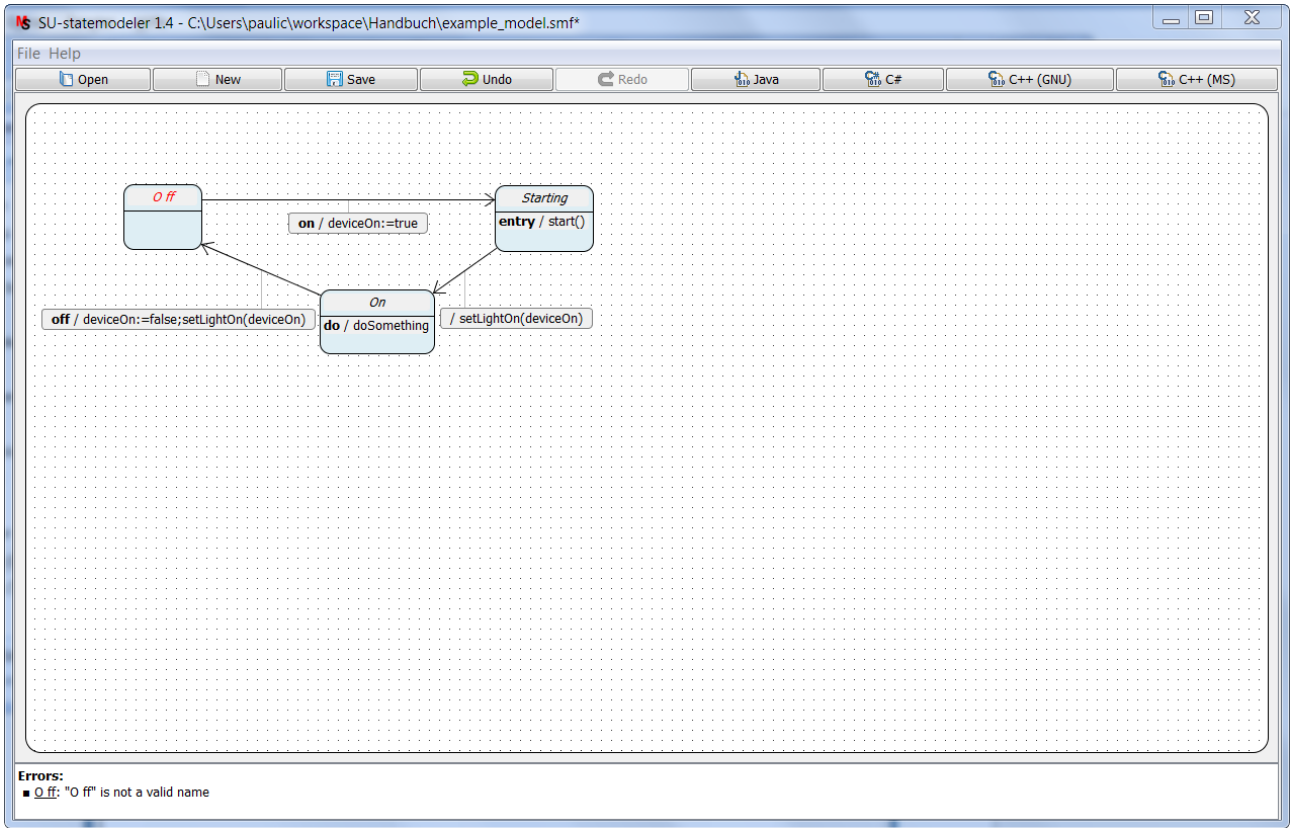
SU-statemodeler can be used as a standalone editor or as a plug-in for Eclipse. The standalone editor only offers a simple user interface and the key features. The more comfortable way to use *SU-statemodeler* is using the plug-in for Eclipse.

5.1 Standalone editor

5.1.1 Overview

The standalone editor offers a simple user interface. Beginning at the top of the window the user interface consists of:

- menu bar
- button bar
- editor panel
- error panel



Picture 29: standalone editor

The menu bar offers some features which are usually less frequently used, e. g. open the manual. Most features are available via buttons of the button bar. Every button offers a short help when moving the cursor on it. The editor panel contains the editor for UML-statecharts and is used as described in chapter 3. The error panel at the bottom shows errors as they are recognized by *SU-statemodeler*. The label **Errors:** is always shown, every following line refers to one error in the model. For every error a short description is shown. In the model as shown in picture 29 the name of the state `O ff` does not comply with the syntax rules.

The title bar of the application shows the path of the currently edited file. The appearance of the application may differ slightly on various platforms. The graphical representation of the objects in the editor can be changed concerning the colors.

5.1.2 System requirements

To run the standalone editor, a Java runtime (version at least 1.7) must be installed. No further libraries are required. The application was successfully tested on Windows XP, 7 and Ubuntu 12.04. The application is launched via a double-click on the Java-archive or the command *java -jar statemodeler.jar [model-file]*.

5.1.3 Obtaining the software

The software consists of one Java-archive which can be downloaded at www.christianpauli.de. An installation is not necessary.

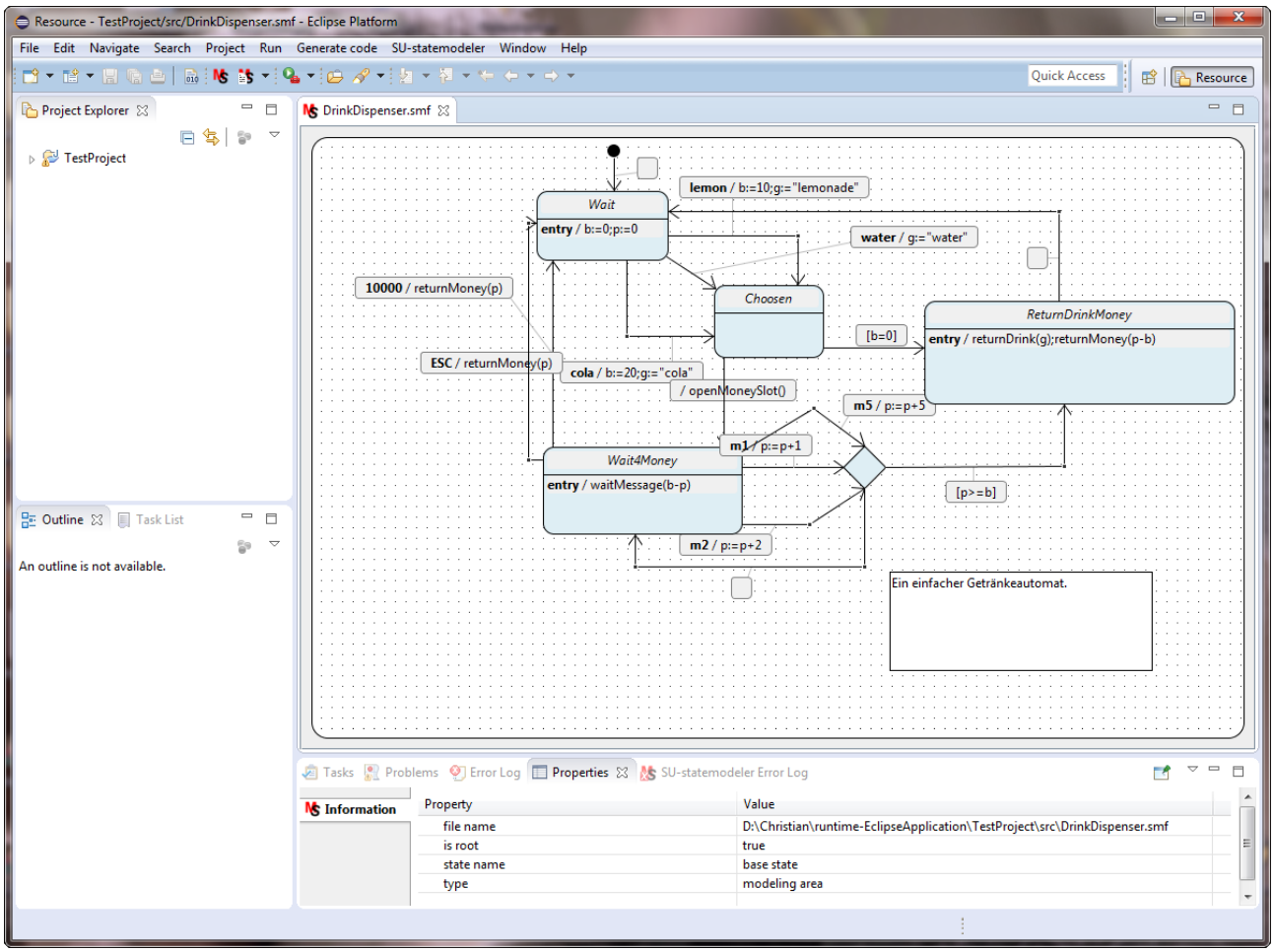
5.2 Eclipse plug-in

5.2.1 Overview

This plug-in offers the user all features of the standalone editor and additional features. These are:

- integration of model files in workspace
- saving of code code configurations
- exporting images in common image formats
- dialog for creating new model files
- property viewer for model objects

Picture 30 gives an overview over the plug-in. The plug-in needs a Java runtime (version at least 1.7) and was successfully tested on Windows XP, 7 and Ubuntu 12.04. It needs Eclipse with version at least 4.3. Lower versions might work, but are not officially supported.



Picture 30: Eclipse plug-in

5.2.2 Model files

All model files are decorated with the *SU-statemodeler* icon, so that they can easily be found in the package explorer. Model files can be integrated into every kind of project in Eclipse.

5.2.2.1 Creating a model file

To create a model file open the menu item *New / Other...*. There choose *SU-statemodeler file* under *SU-statemodeler*. Then a file name and a directory (relative to the project directory) can be specified for the new model file.

5.2.2.2 Opening a model file

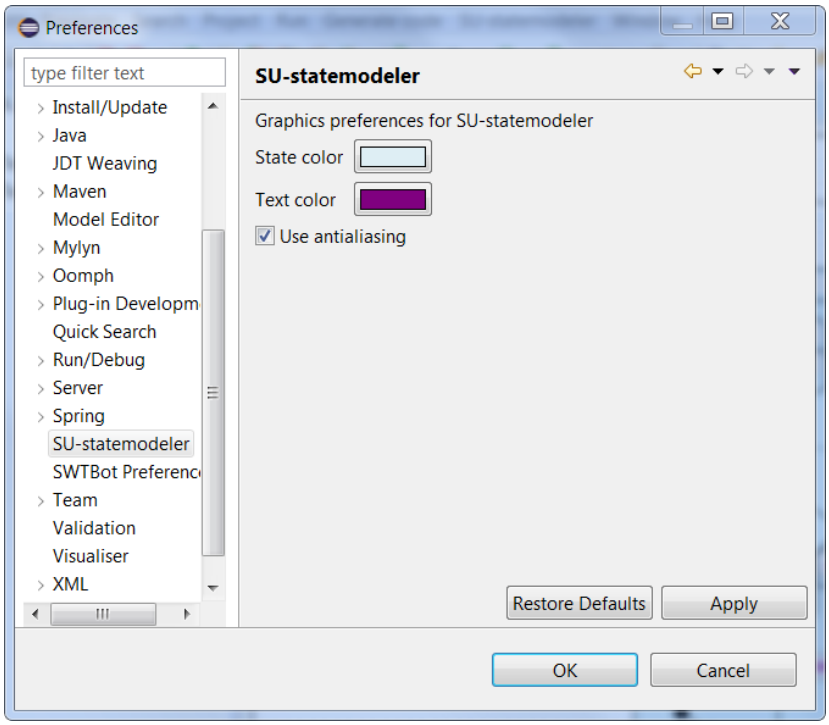
Opening a model file is usually achieved by a double-click on it, because file type *smf* is bound to the editor of *SU-statemodeler*. If this is not the case, then open the context menu of the model file with a single right click on it and choose *Open With / SU-statemodeler*. If you do not use *smf* as file ending for model files, this step might always be necessary. Alternatively it is possible to open and edit model files with a text editor, what is not recommended.

5.2.3 Color preferences

It is possible to change several color-preferences of the editor. The background-color of states and the font-color can be set to an arbitrary color. Furthermore anti-aliasing can be activated for a

prettier presentation of models.

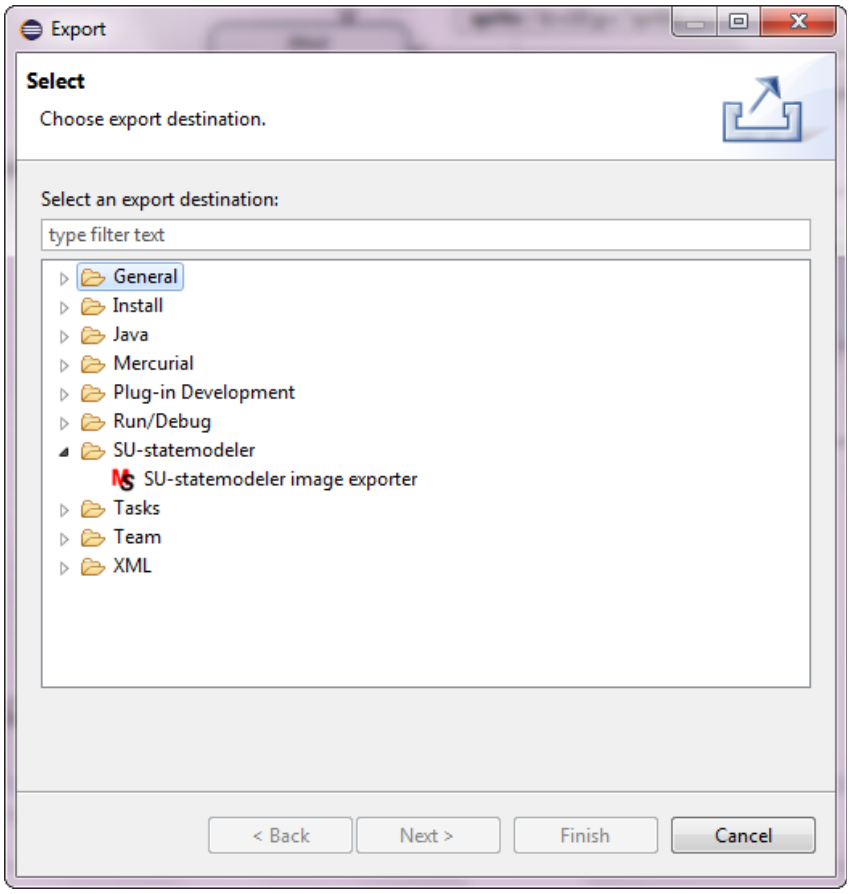
The dialog for changing the settings is accessible via *Preferences / SU-statemodeler*.



Picture 31: Color-preferences

5.2.4 Exporting images

A model can be exported as an image with the image exporter. The image exporter supports common image formats as *png*, *bmp* or *jpg*. The image exporter is opened via *File / Export*. Then image exporter can be found under the item *SU-statemodeler*. After that a file format and name can be chosen.



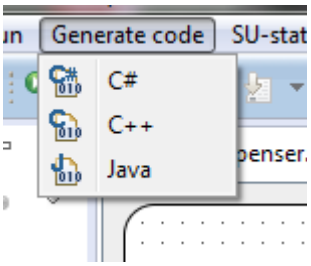
Picture 32: Image exporter

5.2.5 Generating code

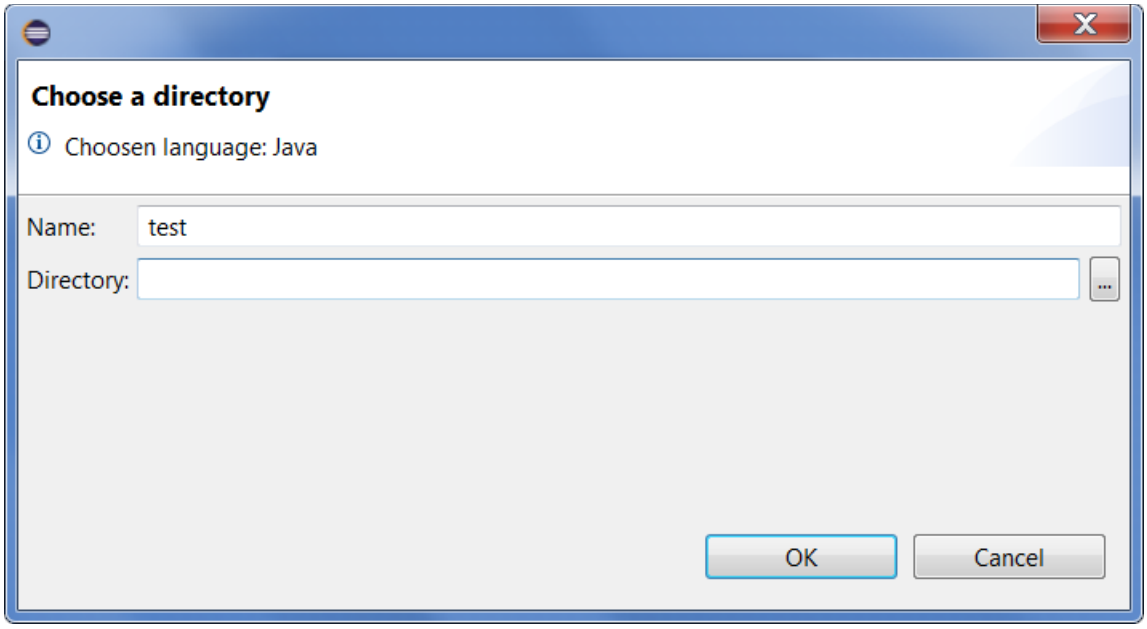
Code is generated via the menu *Generate code* in the menu bar (→ picture 33). Code can only be generated if a model file is opened and the corresponding editor is activated. Otherwise the corresponding menu is disabled. When clicking on a menu item a dialog as shown in picture 34 appears. There a name and a directory can be specified, both together is a *build configuration*. The name is to identify the new build configuration. Afterward this build configuration is stored under the given name and can be carried out again with only one click. Already stored build configurations are visible in the menu.

If the name is left blank, then this build configuration is not stored. If an already chosen name is given, then the old build configuration is overwritten. Build configurations are saved in the workspace and hence bound to it[4]. They can be deleted individually or all at once by clicking the menu item *Remove all configurations* (→ picture 35)

4 stored in XML-file at `.metadata\.plugins\com.statemodeler.plugin`

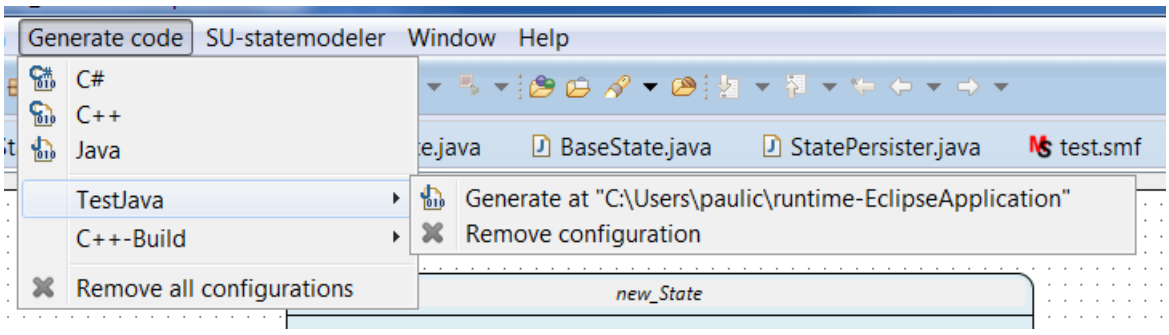


Picture 33: menu generate code

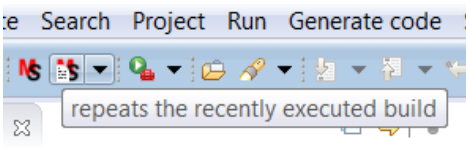


Picture 34: dialog generate code

When generating code for C++ the variant as described in chapter 4.5.7 can be chosen (default is *GNU*). Furthermore a default header included in every generated source file can be chosen.
When clicking the right button shown in picture 36, then the lastly invoked build configuration is repeated again. This is a shortcut. The also available drop-down menu is the same as in picture 33.



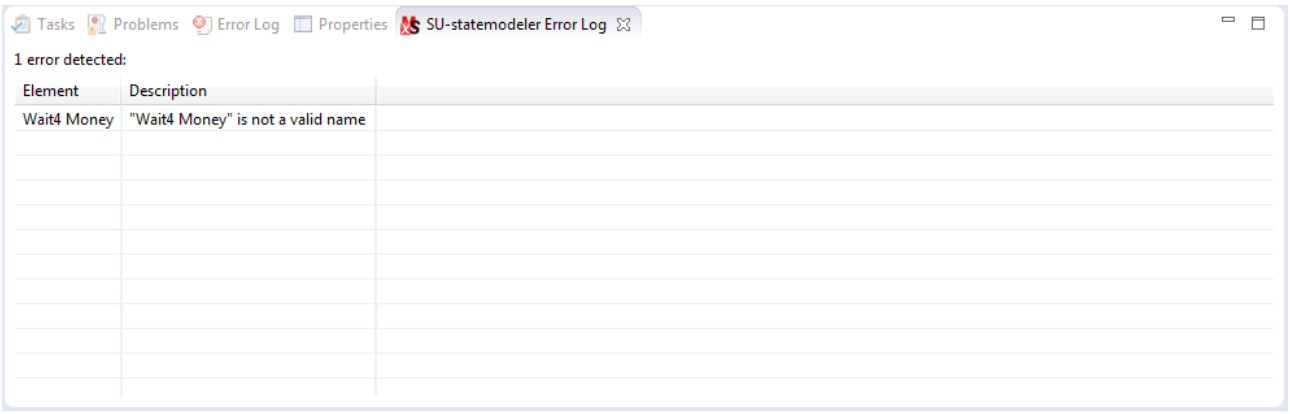
Picture 35: saved build configuration



Picture 36: repeat last build

5.2.6 Error view

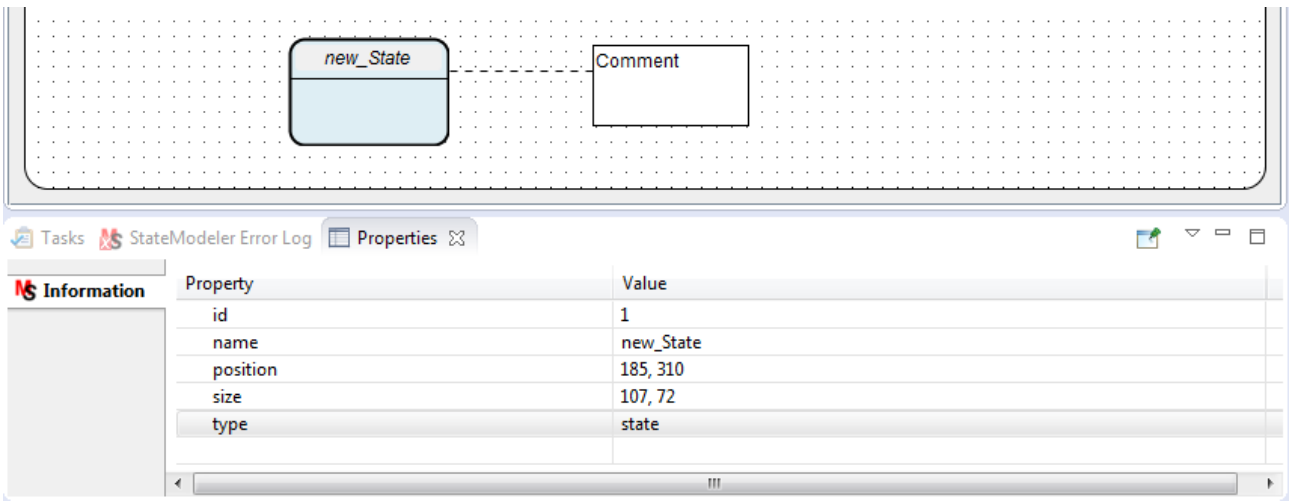
All errors are listed in the *SU-statemodeler Error Log*. This view can be opened under *Window / Show View / Other...*. Then choose *SU-statemodeler Error Log* under *SU-statemodeler*. With a double-click on a list-item the editor jumps the corresponding object.



Picture 37: Error log

5.2.7 Model properties

Eclipse offers a property view. This view is often not visible by default but can be opened via the menu bar. This property view shows all important properties for objects in a model. Picture 38 shows the properties for the state *new_State*.



Picture 38: property view

5.2.8 Obtaining help

The left button shown in picture 36 shows information about the *SU-statemodeler* plug-in. A manual can be opened via *SU-statemodeler / Manual*.

5.2.9 Installing the plug-in

The plug-in can be installed via the Eclipse marketplace or the update-site http://christianpauli.de/su-statemodeler.

List of pictures

Listings

Imprint

Author Christian Pauli

Date November 4, 2018

Contact Mail: christian.pauli@christianpauli.de
Phone: +49 8452 7340689