Christian-Albrechts-Universität zu Kiel

Diploma Thesis

Extension of *KIEL* by Stateflow Charts

cand. inform. Adrian Posor

December 16, 2005

Institute of Computer Science and Applied Mathematics
Real-Time and Embedded Systems Group

Prof. Dr. Reinhard von Hanxleden

Advised by:
Steffen H. Prochnow

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Kiel,

Abstract

The *Kiel Integrated Environment for Layout (KIEL)* application is a tool to create and simulate state diagrams that are dialects of Harel's Statecharts. Special features of *KIEL* are its set of automatic layouters and the simulation environment which uses the concept of dynamic charts. This work extends *KIEL* by the ability to import, export, and simulate *Stateflow* charts. The data structure used by *KIEL* to store state diagrams is described in detail followed by an introduction to *Stateflow* and its API. A comparison of the *KIEL* data structure to the *Stateflow* API objects results in an appropriate mapping between them. This mapping is needed to import and export *Stateflow* charts using the API. The simulation of *Stateflow* charts within *KIEL* is realized with the help of *Stateflow*, which performs all necessary calculations.

Keywords *Statechart*, import, export, state diagram, simulation, *Stateflow*, *Matlab*, *KIEL*

Contents

List of Tables

List of Figures

Listing of the abbreviations

API	Application Programming Interface
DFA	Deterministic Finite Automaton
GUI	Graphical User Interface
KIEL	Kiel Integrated Environment for Layout
UML	Unified Modelling Language

1. Introduction

The majority of computers in use are embedded into electronic devices that they control. They are called *embedded systems*. The development of software for these *embedded systems* is often not realized using typical programming languages like *C* or *Java*. Instead of this, their behavior is described using a more abstract way: a system model. The system model is used to generate program code. This way, the model does not only serve as a specification language, but also as a graphical programming language. Several modeling tools exist on the market that implement a dialect of the so called *Statecharts*.

1.1. The *KIEL* Project

The ***K**iel **I**ntegrated **E**nvironment for **L**ayout* program [11] is a modeling tool for *Statecharts* and has been developed at the chair of *Embedded and Real-Time Systems*. Its main goal is to provide automatic layout capabilities. Several layouters are integrated into this modeling tool. With their help the developer can focus on the structure of the chart and leave the layout in the hands of the computer.

The tool provides a new approach of showing simulation results. States that are not active are collapsed during the simulation to hide the details of the chart that are not important in the current simulation step. At each simulation step a new view of the chart is shown with the currently active states expanded and the non-active states collapsed. This dynamic view helps to overlook the chart during the simulation.

1.2. The task within the scope of the *KIEL* Project

The given task for this diploma thesis is to expand the *KIEL* application by the ability to load and to simulate charts that have been created with *Matlab/Simulink/Stateflow*. Due to the fact that *Matlab/Simulink/Stateflow* has got an API the solution can be reduced to the communication with *Matlab/Simulink/Stateflow*, which performs all the real work for *KIEL*. The requirements are described in detail in Section 3.1.

1.3. Overview

The following chapter (Chapter 2) provides the theoratical background. It deals with deterministic finite automata, state diagrams in general, the internal data structure that is used by *KIEL* to store the chart in memory (Section 2.2.1), and the *Stateflow* [7] modeling tool by *The Mathworks* [13] (Section 2.2.2). The next chapter (Chapter 3) provides detailed information on the new features that have been added to the *KIEL* application as well as the implementation of these features (Section 3.6).

2. Theoretical Basis

This chapter provides the theoratical background behind the system models this document is about. It describes the internal data structure of the *KIEL* application and introduces the modeling tool *Stateflow* along with its API.

The models used in software engineering to model the behavior of a system are based on finite state machines. A state diagram is a graphical representation of a finite state machine. There are many forms of state diagrams that differ slightly in outlook and semantics. One of the most popular form of state diagram is the so-called *Statechart*, which has been developed in 1987 by David Harel [2]. A variant of *Statecharts* has become a part of UML [8].

2.1. Automata

An automaton is a mathematical model for a finite state machine. Formally, a deterministic finite automaton (DFA) is a 5-tuple $(Q, \Sigma, \delta, S_0, F)$, where

- Q denotes a finite set of states.
- Σ denotes a finite set of symbols, called the *alphabet*.
- δ denotes the transition function $\delta : Q \times \Sigma \to Q$.
- $S_0 \in Q$ denotes the start state.
- $F \subset Q$ denotes a set of final states.

Let $\hat{\sigma} : Q \times \Sigma^* \to Q$ be the extended transition function which takes a string of symbols as an argument and returns the state in which the automaton will stay after processing the input. The language L accepted by the deterministic finite automaton is

$$L = \{w \in \Sigma^* | \hat{\sigma}(S_0, w) \in F\}$$

2.2. State Diagram

State diagrams are used to represent graphically finite state machines. State transition tables are another possible representation. A directed graph is a classic form of a state diagram. Each edge in the graph represents a transition between two states and each vertex represents a state. Extensions to this state diagram type, and therefore to the state machine itself, do exist. They extend the diagram /the machine by the concept of hierarchy, parallelism, events, actions and others.

A state diagram consists of three basic types of elements: states, pseudostates, and transitions.

State: A state represents a mode of operation. An invariant condition holds as long as the machine resides in a particular state. If the machine is in a particular state, this state is called active, otherwise it is called inactive. If a state contains states, it is called a composite state, otherwise it is called a simple state. A state consists of one ore more so-called regions. The states in a region are mutually exclusive while the regions themselfs are executed in parallel.

Pseudostate: A pseudostate is a kind of state the machine never resides in. Such a state is neither active nor inactive. The following types of pseudostates generally exist: *initial*, *deepHistory*, *shallowHistory*, *join*, *fork*, *junction*, *choice*, and *terminate*.

- An *initial* pseudostate is the source for a transition to the default state of a composite state. There cannot be more than one initial state in a region.
- A *shallowHistory* pseudostate stores the activity history within a composite state. If the associated composite state is reentered, then the previously active state within this composite state will be activated again. The shallowHistory object only stores the activity history of the level of containment it is placed in.
- A *deepHistory* pseudostate basically works like a shallowHistory object but it also stores the activity history of all states which are children of the states in the level of containment this object is placed in.
- A *join* pseudostate takes two or more concurrent transitions (transitions that originate from concurrent states) as its input and one transition as its output.
- A *fork* pseudostate takes one transition as its input and two or more transitions as its output. It splits a transition into several ones. The execution of the outgoing transitions runs concurrently.
- A *choice* pseudostate, which is used with transitions, provides a decision point in the flow of control. In the UML, it realizes a dynamic conditional branch. The transition actions of the transitions that connect to a choice object are executed directly while the transition segment is taken and before the entire transition is taken. This way the transition actions effect the path which is taken.
- A *junction* pseudostate, like a choice object, provides a decision point in the flow of control. In the UML, it realizes a static conditional branch.

- A *SynchState* is used in conjunction with forks and joins to ensure that one region leaves a particular state or states before another region can enter a particular state or states.
- If a *terminate* pseudostate is entered, then the state machine terminates.

Transition: A transition connects states and pseudo states. Transitions specify which states will become active in the next execution step of the machine. A compound transition is an acyclical chain of transitions joined by pseudostates (except for initial states). If a compound transition is taken, the source state becomes inactive and the destination state becomes active. A transition that connects two states is a special case of a compound transition. Associated with a transition is a trigger, a guard, and an effect.

In Safe State Machines, the special transition types *strong abortion*, *weak abortion*, and *normal termination* exist.

- A *strong abortion* transition, if it is taken, aborts any other ongoing activity. It aborts a state process without letting it perform its reaction in the current instance.
- A *weak abortion* transition lets the aborted state execute its current reaction for the last time, then prevents it from reacting to future instances.
- If all states contained in a composite state have reached a final state then this composite state can be leaved through a *normal termination* transition without the presence of any events.

2.2.1. The *KIEL* Data Structure

This section describes the internal data structure of the *KIEL* application. The data structure is used to represent a chart in memory.

The data structure is designed to support all variants of state diagrams that are similar to those defined in the UML standard. It has been developed using the principles of object oriented programming. The graphical information of a chart is strictly separated from its structure. This is the key to the ability of providing several layouts of the same chart. The following text describes all classes the data structure consists of. The semantics of the elements are described in Section 2.2.

Some classes do just exist to satisfy the needs of *Esterel Studio* [1] charts. On the other hand, some modifications had to be made to support *Stateflow* [7] charts. Figure 2.1 shows a class diagram which gives an overview of the data structure. Figures 2.2, 2.3, and 2.4 provide further details not shown in Figure 2.1. A complete documentation of all fields and methods of these classes can be found in the JavaDoc documentation of the *KIEL* project [12].

StateChart: An instance of this class holds the complete chart. It stores information about the type of chart, the input and output events as well as the input and output variables. The field `rootNode` contains the state that contains all other states of the chart.

State: An instance of this class represents a state. It is the base class for all kinds of states and usually not used directly. It stores the actions and variables which are associated with a state. Its direct subclasses are: `SimpleState`, `CompositeState`, and `FinalState`.

SimpleState: This class does not add any functionality to its base class. It just exists to have a type for a simple state, so that the base class does not need to be used directly.

FinalState: An instance of this class represents the so called final state. If the execution of the diagram reaches this state, then the chart will terminate.

CompositeState: This class is the base class for all states that can contain other states. Direct subclasses are `ANDState`, `ORState`, and `Region`.

ANDState: An instance of this class represents a state which can contain other states that run concurrently. The states contained in such a state are of type `Region`.

ORState: An instance of this class represents a state that can contain other states that do not run concurrently. Only one state included in an `ORState` can be active at an instance of time.

Region: An instance of this class represents a state that is contained in an AND-State and runs concurrently with the other states that are at the same level of hierarchy.

PseudoState: This class is the base class for all pseudo states. Direct subclasses are: `Choice`, `DeepHistory`, `DynamicChoice`, `ForkConnector`, `History`, `InitialState`, `Join`, `Junction`, `Suspend`, and `SynchState`.

Choice: An instance of this class represents a choice object. In the UML, it realizes a dynamic conditional branch, in *Esterel Studio*, it realizes a static conditional branch.

DynamicChoice: An instance of this class represents a dynamic choice object. It realizes a dynamic conditional branch.

History: An instance of this class represents a (shallow) history object.

DeepHistory: An instance of this class represents a deep history object.

InitialState: An instance of this class represents an initial state object. Such an object only appears together with an initial arc. The initial arc points to the state within an `ORState` which is set to be active when the `ORState` is entered.

Join: An instance of this class represents a join object.

Junction: An instance of this class represents a junction object.

SuspendState: An instance of this class represents a suspend state. It is always the destination of a suspension transition. If such a state is active, then the chart is suspended.

SynchState: An instance of this class represents a SynchState object. Such an object is a pseudo state.

ForkConnector: An instance of this class represents a fork connector object.

Transition: This class is the base class for all types of transitions. It contains a `Label` and a `Priority`. Direct subclasses are: `ConditionalTransition`, `InitialArc`, `NormalTermination`, `StrongAbortion`, `Suspension`, and `Weak-Abortion`.

ConditionalTransition: An instance of this class represents a transition that is associated with a condition that has to be satisfied to take this transition. This class does not add any functionality to its base class. It just exists to have a type for conditional transitions.

InitalArc: An instance of this class represents an initial arc object. Such an object is a special kind of transition. Its source is always an initial state. Its destination is the state that will be activated when its parent, an ORState, is entered.

NormalTermination: An instance of this class represents a special kind of transition that is called Normal-Termination.

StrongAbortion: An instance of this class represents a special kind of transition that is called Strong-Abortion.

WeakAbortion: An instance of this class represents a Weak-Abortion transition.

Suspension: An instance of this class represents a supsension transition. The destination of such a transition is always a suspend state. If the events associated with a suspension transition are present, then the transition will be taken and the chart will be suspended.

Edge: An instance of this class represents an edge. An edge always links two nodes. It stores a source and a destination. Direct subclass is: `Transition`.

Node: An instance of this class represents a node. It stores its parent, all incoming and outgoing edges/transitions and its name. Direct subclasses are: `PseudoState`, and `State`.

GraphicalObject: All objects that have a graphical representation are derived from this class. This class is not used directly. It contains an unique identifier. Direct subclasses are: `DelimiterLine`, `Edge`, `Node`, `Priority`, and `TransitionLabel`.

DelimiterLine: An instance of this class represents a line between two regions. It does not contain any graphical information like the position of the line.

Priority: Every transition has a priority that is represented by this class. It contains a value that is the value of the priority.

TransitionLabel: This is the base class for all kinds of transition labels. Direct subclasses are: `CompoundLabel`, `StringLabel`.

CompoundLabel: An instance of this class represents a label that is composed of a condition, an effect, and a trigger.

StringLabel: An instance of this class represents a transition label as a string. The string has to be parsed to split it into a condition, an effect, and a trigger. The `CompoundLabel` is the parsed form of the `Stringlabel`.

Variable: This class is the base class for all kinds of variables. Currently, variables of type integer (32-bit, 16-bit, 8-bit), double, float, boolean and string are supported. Direct subclasses are: `StringVariable`, `IntegerVariable`, `Integer16Variable`, `UInt32Variable`, `UInt16Variable`, `UInt8Variable`, `Integer8Variable` `DoubleVariable`, `FloatVariable`, and `BooleanVariable`.

Constant: This class is the base class for all kinds of constants. Currently, constants of type integer (32-bit, 16-bit, 8-bit), unsigned integer (32-bit, 16-bit, 8-bit), double, float, boolean, and string are supported. Direct subclasses are: `StringConstant`, `IntegerConstant`, `Integer16Constant`, `Integer8Constant`, `UInt32Constant`, `UInt16Constant`, `UInt8Constant` `DoubleConstant`, `FloatConstant`, `BooleanFalse`, and `BooleanTrue`.

2.2.2. Stateflow Objects

Stateflow is a modeling tool by *The Mathworks* [13]. It borrows concepts from *Statecharts* which have been first introduced by David Harel [2]. Therefore, you can treat *Stateflow* as a *Statechart* variant. Figure 2.5 shows the *Stateflow* application window. The following text describes *Stateflow* charts in detail.

Stateflow diagrams are embedded into a *Simulink* environment which serves as a source of input and a sink of output to the diagram. The output of a diagram controls a physical plant modeled in *Simulink*. A *Stateflow* diagram represents an object with discrete modes denoted as *states*. It reacts to events by switching from one state to another. The behavior of the object modeled by the diagram depends

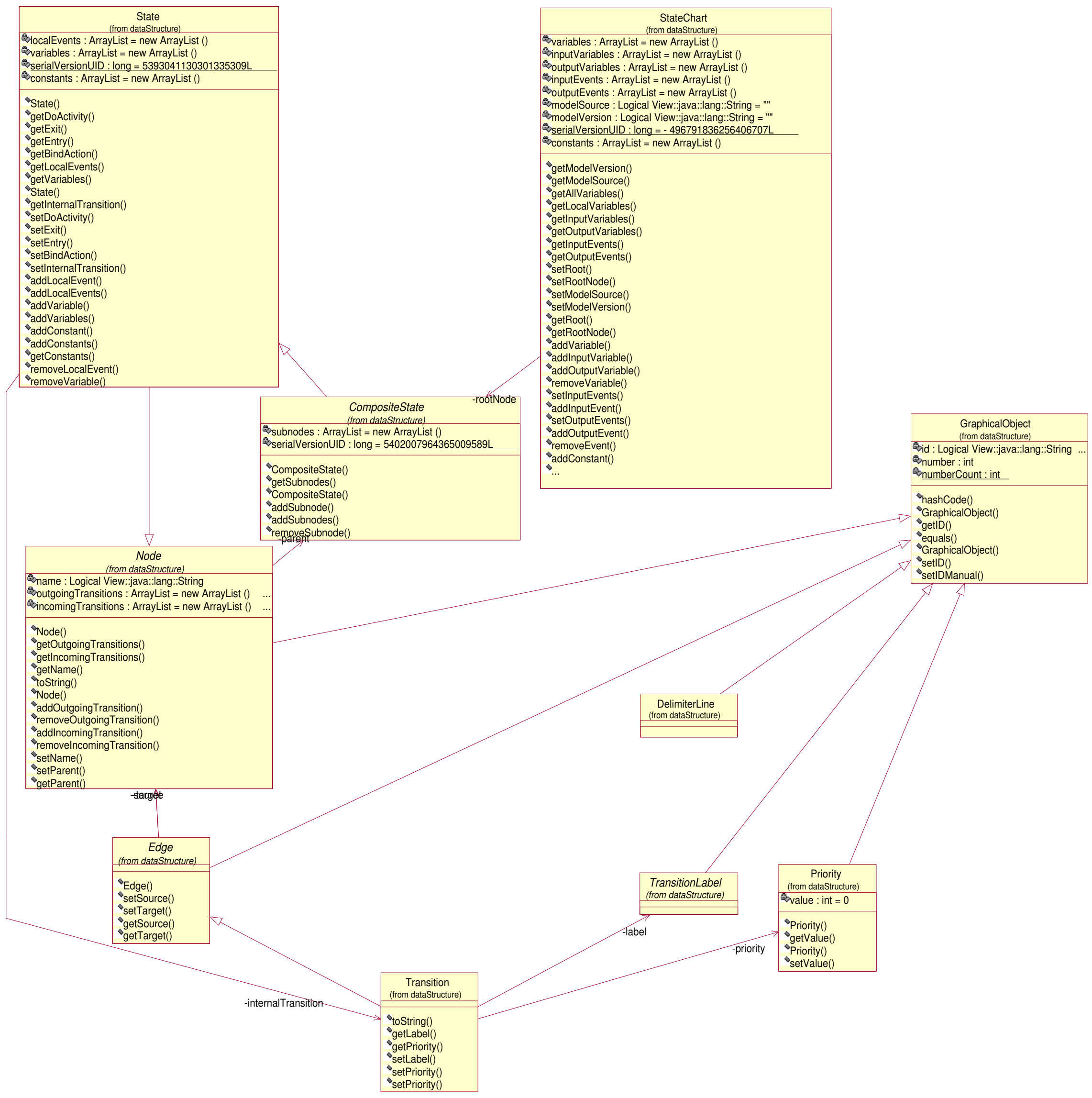


Figure 2.1.: The *KIEL* data structure class diagram

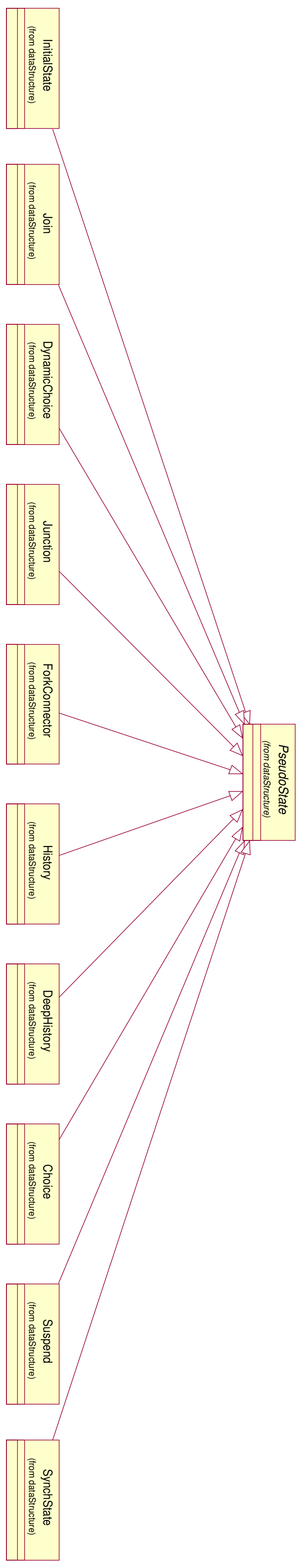


Figure 2.2.: PseudoState and its subclasses

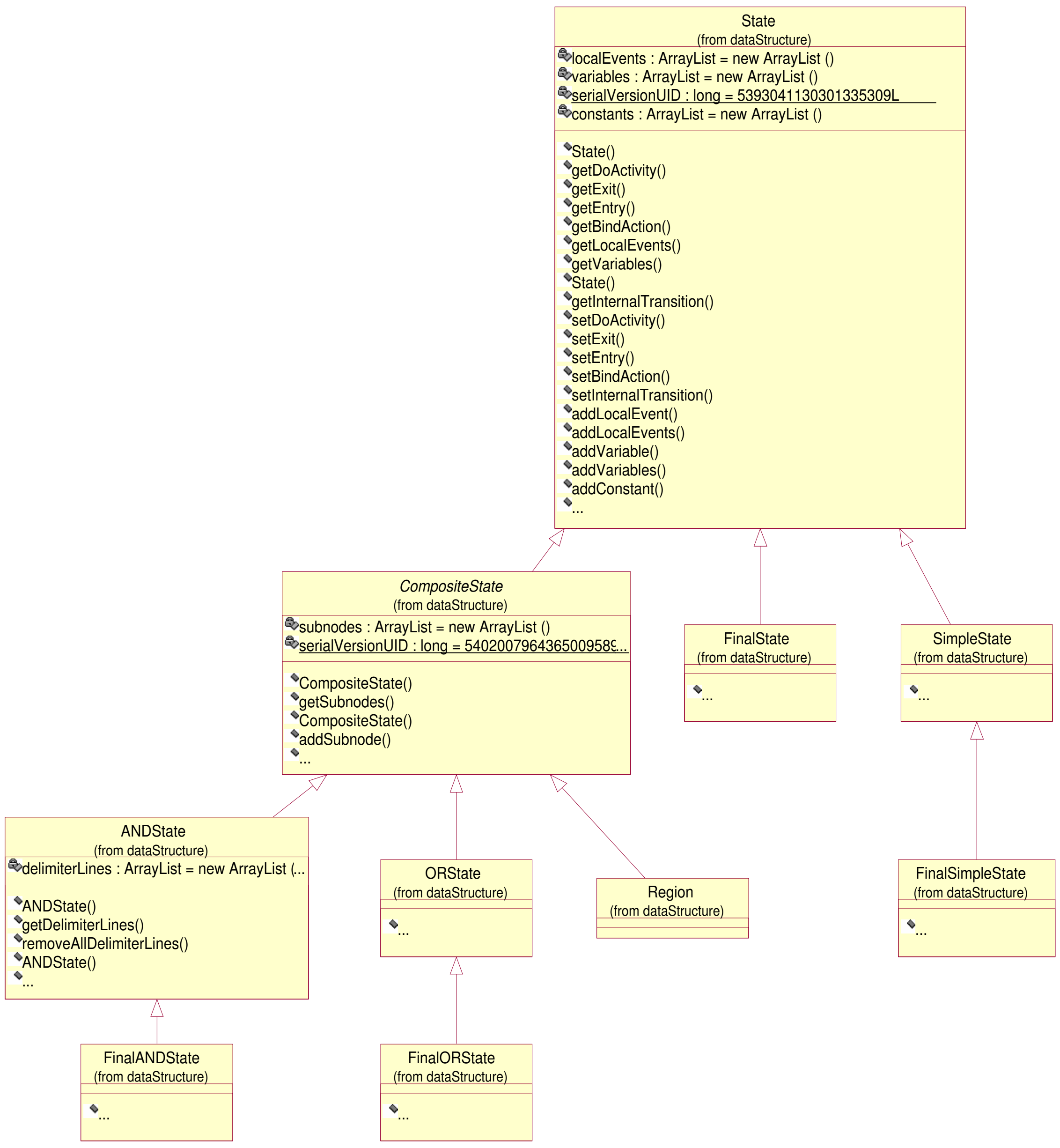


Figure 2.3.: State and its subclasses

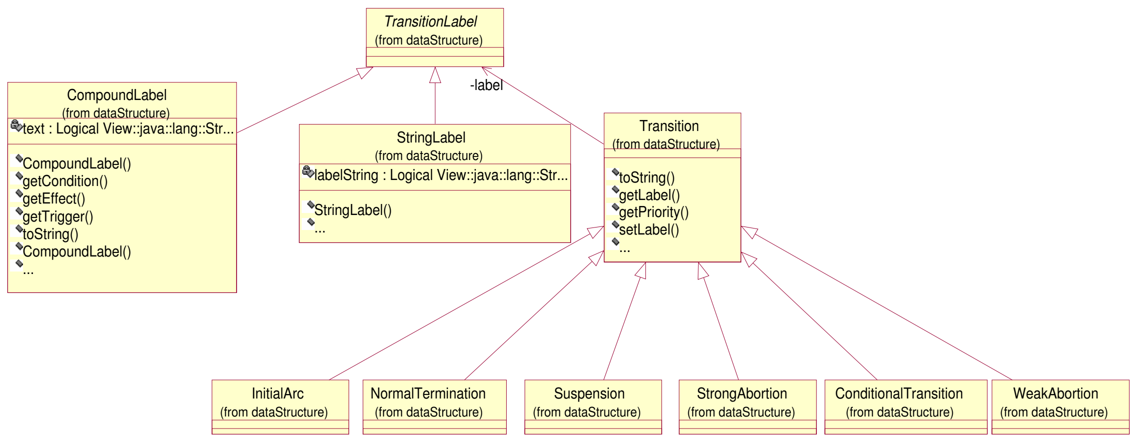


Figure 2.4.: Transition and its subclasses

on the current state the diagram is in and on how the diagram switches from one state to another.

A *Stateflow* diagram consists of:

- A set of graphical objects
- A set of non-graphical objects
- Defined relationships between those objects

Figure 2.6 shows all *Stateflow* objects and their hierarchy. Table 2.1 shows the notation of the graphical objects.

States

A state describes a particular mode of the system. It can be either active or inactive. The switch between active and inactive takes place due to events and conditions. In other words, the execution of the diagram is driven by the occurrence of events, which make states become active or inactive. At any instance of time, there is a set of active states. Every state has a parent, that means it is contained by another state. This is the concept of hierarchy. The diagram itself is called the Stateflow diagram root and it contains all other states. The activity history of a state can be recorded in a *history junction*, which makes it possible to make future activity depend on past activity. States have labels that can specify actions, which are executed in sequence according to the type of any particular action. The supported types of actions are *entry*, *during*, *exit*, *on* `event_name`, and *bind*.

The entry actions of a state are executed when the state becomes active. Its exit actions are executed when the state becomes inactive. Its during actions are executed when the state is active, an event occurs, and no transition out of that state can be taken. Data and events associated with a bind keyword can only be changed by actions of the state and its children they belong to. Other states can

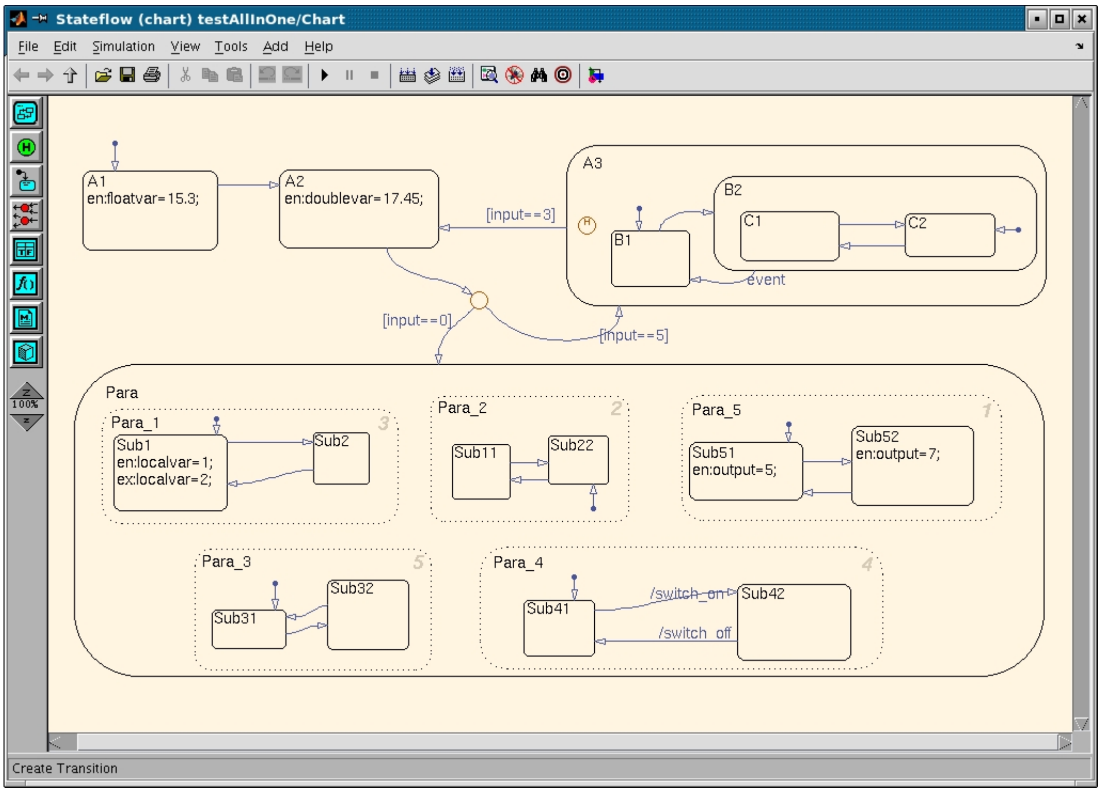


Figure 2.5.: A Stateflow window showing a diagram

only read the bounded data. The `on` *`event_name`* actions are executed when the state they belong to is active and the event *`event_name`* occurs.

To support parallelism *Stateflow* distinguishes between the two *decomposition* types “parallel and” and “exclusive or”. A state of decomposition type “exclusive or” contains exclusive (OR) states. Only one of the OR states can be active at an instance of time. On the other hand, a state of decomposition type “parallel and” contains parallel (AND) states. All of the AND states at the same level in the hierarchy are either active or not active at the same time. AND states are drawn as dashed rectangles, and they are independent of each other.

States can be classified as superstates, substates, or just as states. A state is a superstate if it contains other states, which are called substates. A state that is neither a superstate nor a substate is just a state.

The label of a state has to be of the following format:

```
name/
entry:entry action
during:during action
exit:exit action
on event_name:on event_name action
bind:events, data
```

The label always begins with the name of the state followed by an optional "/" character. The lines that follow the name are keyword prefixes that identify particular types of actions. The supported types of actions have already been presented above. Each keyword prefix is followed by an action statement. Multiple actions for each action type have to be separated either by a carriage return, a semicolon, or a comma. There can be multiple `on` *`event_name`* lines thus specifying multiple `on` *`event_name`* actions for different events. Each keyword is optional, and the keywords do not have to appear in a particular order. The action statements are a part of the *Stateflow* action language.

Transitions

A transition usually links one object to another. It has a source, a destination, and a label. The transition label defines the circumstances under which the associated transition is taken. To take a transition means to leave the state that is the source of the transition and to enter the state that is the destination. The occurrence of an event together with the satisfaction of a particular condition causes a transition to be taken. Transitions are drawn as curved lines with an arrowhead at the destination side.

Default transitions specify which OR state within a superstate of decomposition type "exclusive or" has to be entered when the superstate is entered.

The label of a transition has to be in the following format:
`event[condition]{condition_action}/transition_action` The event named in the label causes the transition to be taken if the event is present and the given condition is true. If no event is named in the label, then the transition will react to any event. It is possible to specify multiple events in the label by using the logical OR operator. The condition in the label is a boolean expression. If the condition evaluates to true and the named event in the label is present, then the associated transition will be valid to be taken. The condition action, which follows the condition and is enclosed in curly braces, is executed as soon as the condition is evaluated to true and right before the destination of the transition has been determined to be valid. In case no condition is specified, the condition action will be executed anyway. The transition action is executed right before the destination state has been entered.

Events

Events are non-graphical objects, which drive the execution of a diagram. If an event occurs, then the status of every state in the diagram will be evaluated. This can result in a transition to be taken or an action to be executed. Events can be created at any level in the hierarchy of the diagram. They are visible to the state they belong to and to its children only. An event is present for one calculation step only. After all new active states have been determined, all events are absent again.

An event can be:

- Local to the *Stateflow* diagram

- An input to the *Stateflow* diagram from its *Simulink* model
- An output to its *Simulink* model from the *Stateflow* diagram
- Exported to a code destination that is external to the diagram and the *Simulink* model
- Imported from a code source which is external to the diagram and the *Simulink* model

The last two points are not relevant for this work.

Data

Data objects are variables that hold numerical values. They are non-graphical objects.

A data object can be:

- Local to the *Stateflow* diagram
- An input to the *Stateflow* diagram from its *Simulink* model
- An output to its *Simulink* model from the *Stateflow* diagram
- Non-persistent temporary data
- Defined in the Matlab workspace
- A constant
- Exported to a code destination that is external to the diagram and the *Simulink* model
- Imported from a code source that is external to the diagram and the *Simulink* model

The last two points are not relevant for this work.

Connective Junctions

Connective junctions are graphical objects, which are used together with transitions to provide a control flow construct. They are displayed as small circles. A connective junction divides a complete transition into transition segments. Each connective junction is a decision point between alternate transition paths. Connective junctions can be used to construct flow diagrams. That are diagrams which consist of transitions and connective junctions only. It is important to notice that an event cannot trigger a transition from a connective junction to a destination state.

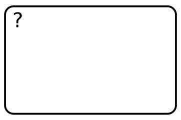
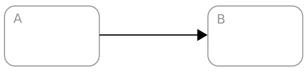
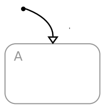
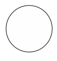
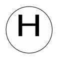
Object Name	**Graphical Notation**
State	
Transition	
Default Transition	
Junction	
History	

Table 2.1.: Notation of *Stateflow* Objects

History Junctions

A history junction records the most recently active state of a superstate. If a superstate containing a history junction is entered, then the state that had been active as the superstate became inactive before will become active again. In case the superstate is entered for the first time, the active state will be determined by the default transition. Please note that the history junction only applies to the level of the hierarchy in which it appears.

Actions

Actions appear in transition and state labels. Their execution is part of the diagram execution. Transition actions are executed when the associated transition is taken. Actions in states are executed according to their type as described in Section 2.2.2. An action can be a function call, the broadcast of an event, the assignment of a value to a variable, and so on. The operators that are available in the action language are similar to the operators known from high level programming languages like C. Among these are Multiplication, Division, Addition, Substraction, bitwise operators, logical operators, Assignment, pointer and address operators. There are also temporal logic operators, which is something usually not found in programming languages. For example, the operator `after` takes an event and an expression `n` that evaluates to a positive integer value. The operator evaluates to `true` if the given event has occurred `n` times.

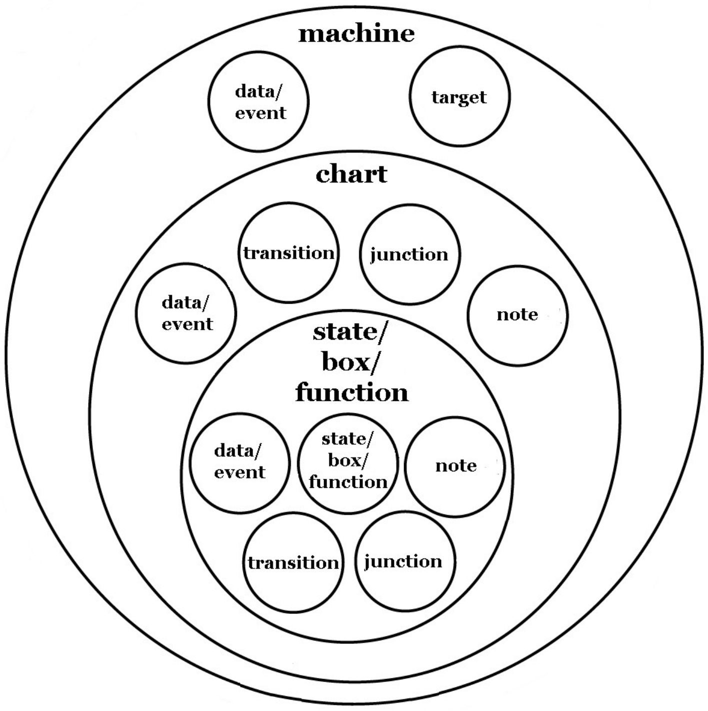


Figure 2.6.: Stateflow object hierarchy (Source: Stateflow User's Guide [7])

2.3. Matlab/Stateflow-API

Matlab/Simulink/Stateflow provides a set of commands that let the user load, save, create, and manipulate state diagrams, define settings, start a simulation, retrieve simulation results. Nearly everything that can be done using the GUI can also be done using Matlab's commands. The user can enter these commands at the Matlab prompt. It is possible to write these commands into a file, so that they form a script. Scripts automate the work like the editing of a diagram. They can be invoked within the command line like the build-in commands. Although *The Mathworks* calls this set of commands an API, it is not designed to be used within a programming language. However, as long as it is possible to attach Matlab's input and output streams to a process that wants to interact with Matlab, this process can generate Matlab "scripts" on-the-fly. A study of the possibility of a simulation using the API was done by Jan Täubrich [10].

Each object in *Stateflow*, including the graphical and non-graphical objects described in 2.2.2, has a corresponding object in the API. Figure 2.7 shows all API objects and their hierarchy. An API object bundles a set of specific functions and properties. The change of a property of an API object also changes the property of the corresponding *Stateflow* object. The API objects, which correspond to the objects described in Section 2.2.2, have the same hierarchy as their counterparts. There are some objects that do not fit into the hierarchy model like the `Editor` and the `Clipboard` object. The `Editor` object exists to provide access to purely graphical aspects of a chart like the size of the chart in pixel. The `Clipboard` object provides copy and paste functionality. The `Machine` object in figure 2.6 corresponds to the `model` object in the API. This object represents the *Simulink* model, which includes the *Stateflow* diagram. The `Root` object contains all *Stateflow* API objects. It exists to have a means to distinguish *Stateflow* API objects from API objects of other tools like *Simulink*. The `Chart` object represents the *Stateflow* diagram. There can be multiple diagrams within a model.

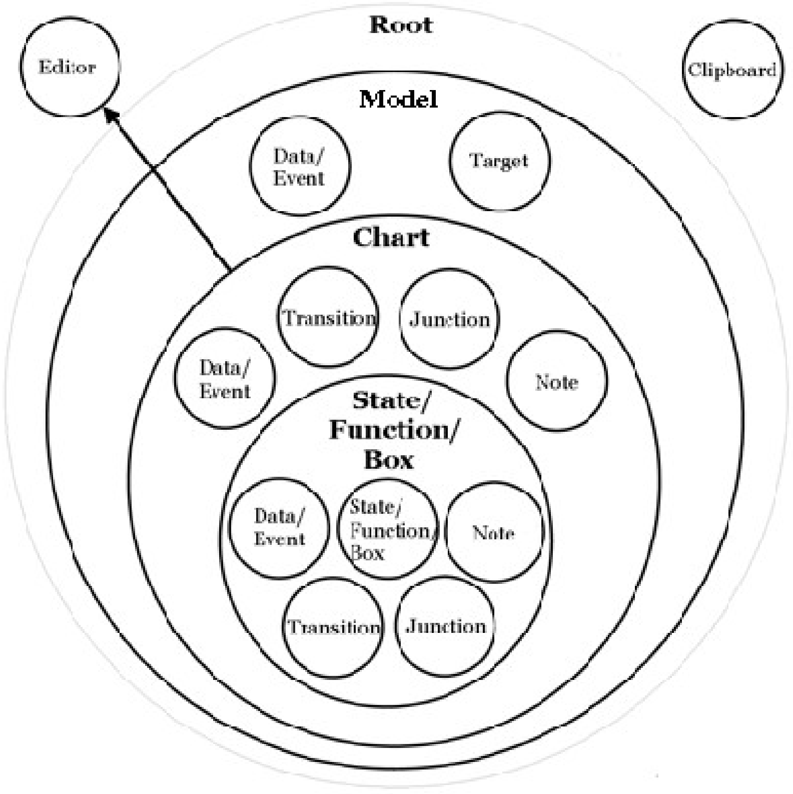


Figure 2.7.: Stateflow API object hierarchy (Source: Stateflow API reference [6])

API objects are accessed through variables that contain references to these objects. Such variables are called *handles*. API object properties correspond to the values one can set using the diagram editor. API object methods provide a means to create, delete, change, and find the object they belong to. All objects have some properties and methods in common. For example, the `Id` and the `Description` property are common properties. On the other hand, there are properties and methods that are specific to the type of object, like the `Source` or `DecompositionType` properties. A complete list of all methods and properties can be found in the *Stateflow* API reference [6].

By convention, all property names start with a capital letter, while all method names start with a letter in lowercase. In names that consist of concatenated words, each word starts with a capital letter.

To access a property or method of an object, the so-called dot notation is used. One writes down the name of the variable that contains the handle to the object followed by a dot character and then the name of the property or method one wants to access. It is allowed to write down nested dot expressions. If a function call needs to return more than one object, it returns an array of objects. To access a specific item in an array, one has to append the index of the item needed enclosed in round braces. Example:

```
label1 = sA1.innerTransitionsOf(1).LabelString
```

Special constructor methods exist to create new objects. They return handles to the newly created objects. The constructor methods used in this work are: `Stateflow.Data`, `Stateflow.Event`, `Stateflow.Junction`, `Stateflow.State`. The created objects have a destructor method called `delete`.

2.4. Comparison of *KIEL* Data Structure classes with *Stateflow* API Objects

To represent a *Stateflow* chart in the *KIEL* application, it is necessary to associate each *Stateflow* API object with an object in the *KIEL* data structure. The Table 2.2 associates the classes of the *KIEL* data structure with the objects of the *Stateflow* API. Not all objects of the *KIEL* data structure do exist in the API and the other way around. As long as no support for graphical functions and truth tables is needed, no new classes need to be created in the *KIEL* data structure (except for the support of variables of type *double* and *float*). `Action` and `Event` do not exist as objects in the *Stateflow* API. They are included as strings in transition labels. The transition labels need to be parsed to create the objects `Action` and `Event`. Because *Stateflow* uses dashed rectangles to represent parallel states instead of delimiter lines, these delimiter lines (represented by `DelimiterLine`) have to be generated during the import of *Stateflow* charts.

KIEL data structure	Stateflow API
StateChart	Stateflow.Chart
ORState	Stateflow.State.Decomposition= ’EXCLUSIVE_OR’ Stateflow.State.Type=’OR’
ANDState	Stateflow.State.Decomposition= ’PARALLEL_AND’ Stateflow.State.Type=’AND’
SimpleState	non-existent, use an ORState
FinalState	non-existent
Final*	non-existent
InitialState	non-existent
PseudoState	non-existent
Junction (not dynamical)	Stateflow.Junction.type= ’CONNECTIVE_JUNCTION’
History (shallow)	Stateflow.Junction.type= ’HISTORY_JUNCTION’ Stateflow.Junction.labelString="H"
DeepHistory, Choice, DynamicChoice, ForkConnector, Join, Suspend, SynchState	non-existent
Transition	Stateflow.Transition
TransitionLabel/StringLabel	Stateflow.Transition.LabelString
CompoundLabel	non-existent
ConditionalTransition	Stateflow.Transition
NormalTermination, StrongAbortion, WeakAbortion	
Suspension	non-existent
Region	non-existent
Signal	non-existent
Variable	Data
StringVariable	non-existent
Event	Stateflow.Events
Actions/Action	non-existent as Obj.; in transition label only
non-existent	Machine, Box, Note, Target
non-existent	GraphicalFunction
non-existent	TruthTable
GraphicalObject	non-existent
Node, Edge	non-existent
DelimiterLine	non-existent; has to be generated
Priority	exists only implicitly

Table 2.2.: Comparison of *KIEL* data structure classes with Stateflow API objects

3. *Stateflow* Extensions of *KIEL*

The previous chapter includes the background information needed to understand what this diploma thesis is about. The following sections introduce the new features that have been added to the *KIEL* application. The features are described from the user's point of view. Details of the implementation are given after it.

3.1. Requirements

The *KIEL* program is supposed to be able to import charts that have been created with *Stateflow* [7] and are stored in model files that have the file name extension "mdl". Charts that make use of one or more of the following features do not have to be supported: interlevel transitions, graphical functions, truth tables, *C*-function calls. Even no support is required for model files that include *Simulink* blocks or more than one *Stateflow* chart. The layout of the imported chart should be as close as possible to that of the original chart. For charts that include subcharts, it is not required to preserve the original layout during import, because the *KIEL* program does not support the concept of subcharts. Furthermore, the program has to be able to simulate *Stateflow* charts. Macro step simulation has to be supported, micro step simulation is not necessary. Simulation results have to be shown in the browser in the same way it is already done for *Esterel Studio* [1] charts. The table shown in the Browser has to be extended by tabs for input variables, output variables, and input events.

Events are a concept of *Stateflow*, while signals are a concept of *Simulink*. Signals trigger events. Simulating charts with *Stateflow*, the user specifies the values of the input signals for each instance of time. The change in value of a signal triggers the corresponding event either in case a falling edge or a rising edge occurs, or in both cases according to the type of the trigger. In *KIEL* the possibility of specifying events instead of signals has to be added to provide two different kind of views of the simulation input. *KIEL* has to support the file formats used by *Stateflow* and *Esterel Studio* to store the simulation input data. All simulation input data entered by the user has to be stored and the possibility to save this data in a format that *Stateflow* or *Esterel Studio* can read has to be provided. This is the input trace feature and it is described in detail in Section 3.3.

The editor included in the *KIEL* application is designed for the creation of *Esterel Studio* charts. It provides a special mode to support the creation of *Stateflow* charts, but this mode is not fully implemented. The editor is supposed to support the creation of *Stateflow* charts that make use of all features not excluded above.

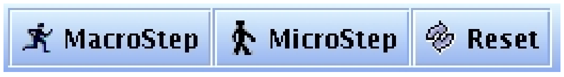


Figure 3.1.: The simulation toolbar of the *KIEL* browser

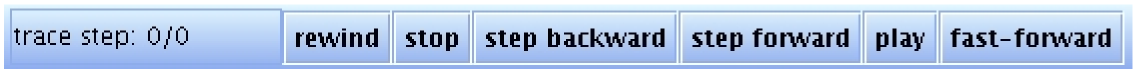


Figure 3.2.: The trace toolbar of the *KIEL* browser

The following sections describe the features that have been added to *KIEL* followed by details of their implementation.

3.2. How to simulate a *Stateflow* Chart

Charts are simulated inside the browser. All variables, signals, and events can be set in the table in the lower half of the browser window. The user can choose to specify either signals or events, but not both at the same time. The difference between signals and events is described in Section 3.1. A click onto `MacroStep` (Figure 3.1) computes the next step according to the current state of the chart and the input data from the table. As a result of a simulation step, all active states are marked, and the table shows the current values of all variables and which events are currently present. If the user selects one of the automatic layouters, then states that are not active are collapsed (interior is not shown) and the layout changes to hide non-interesting parts of the chart. The simulation of micro steps is not available for *Stateflow* charts. Figure 3.1 shows the simulation toolbar of the browser.

3.3. The input trace Feature

The input made by the user at each simulation step is recorded for the purpose of playback at a later date. The recorded input is called the *input trace*. The buttons `rewind`, `stop`, `step backward`, `step forward`, `play` and `fast-forward` (Figure 3.2) have been added to the browser to allow control over the recorded input data. The text field on the left side of the `rewind` button shows the current trace step number and the total number of trace steps that have been recorded. Every time the user clicks onto `MacroStep` (Figure 3.1) the input data (the set of signals that are present and the values of variables) of the current step is added to the trace. The user can replay the simulation steps he has made by rewinding the trace and clicking onto `step forward` to replay one simulation step or onto `play` to replay the simulation from the current step up to the last step. If the user clicks onto `MacroStep` while the current trace step is not the last one, then all steps that follow the current one are deleted and the next step is replaced by the new one. The input trace can be saved in the ESI file format as well as in the *Matlab* M file format.

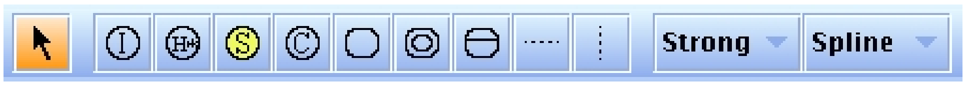


Figure 3.3.: The *Esterel Studio* dependent part of the editor's toolbar

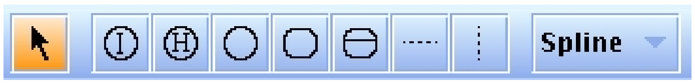


Figure 3.4.: The *Stateflow* dependent part of the editor's toolbar

3.4. Import and Export of *Stateflow* Charts

The user can load a *Stateflow* chart by clicking onto "File", and then clicking onto "Open..". In the following dialog the user has to choose a file with the extension "mdl". The user can specify "Matlab/Stateflow" in the file filter box to see "mdl" files only. To save a *Stateflow* chart the user has to click onto "Save" or "Save as.." in the "File" menu. For details on the implementation read Sections 3.6.6 and 3.6.7.

3.5. *Stateflow* Mode of the Editor

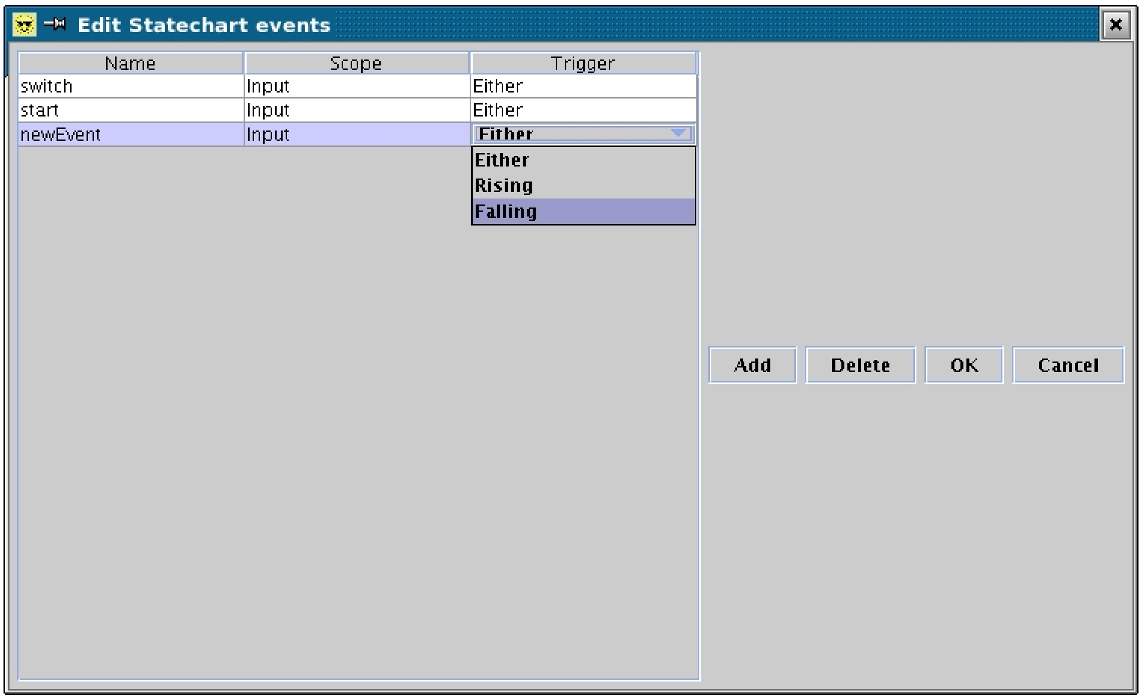
The editor supports the creation of *Esterel Studio* charts as well as of *Stateflow* charts. In dependence of the type of the chart currently edited, the editor shows one of two different toolbars (Figure 3.3 and 3.4). The tools used to create a deep history, a suspend state, a final state, and a dynamic choice are not present in *Stateflow* mode. There is a tool for the creation of a junction instead. The context menus are also different. For example, there is only one type of transition in *Stateflow*,so so you cannot convert a transition to a weak, a strong, or a normal one. The dialogs that allow the editing of variables and of events have been re-designed (Figure 3.6 and 3.5). Most of the data types available in *Stateflow* are supported, and it is possible to set the trigger type of events as well as the scope of variables and of events.

3.6. Implementation

This section focuses on the implementation details of the added features described before. The overall structure of the *KIEL* application is presented. Details on the specific usage of the *Stateflow* API within *KIEL* are covered, and the algorithms used to import and export charts are described.

3.6.1. Surroundings

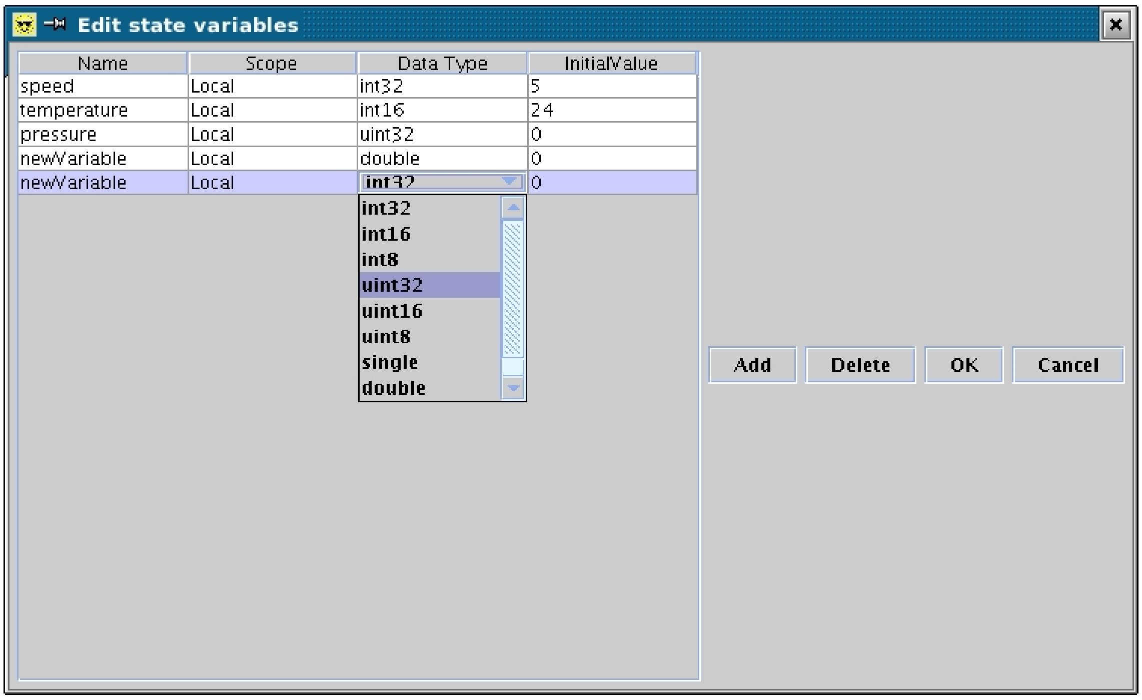
The *KIEL* program is written in *Java*, Version 1.4.2. An advantage of this language is its independence of any particular operating system. Care has been taken not to



Edit Statechart events

Name	Scope	Trigger
switch	Input	Either
start	Input	Either
newEvent	Input	Either

Figure 3.5.: The editor's dialog for editing chart events



Edit state variables

Name	Scope	Data Type	InitialValue
speed	Local	int32	5
temperature	Local	int16	24
pressure	Local	uint32	0
newVariable	Local	double	0
newVariable	Local	int32	0

Figure 3.6.: The editor's dialog for editing state variables

include any operating system specific program code. *KIEL* is regularly tested on SuSE Linux 9.2 and Microsoft Windows XP.

Structure of *KIEL*

The *KIEL* program consists of several independent components, which have been developed in the scope of student projects and diploma thesises. These components communicate through well defined interfaces, which make them easily interchangeable. They all share the *KIEL* data structure, which is described in Section 2.2.1. Figure 3.7 shows the components as well as their comunication paths. The following components do currently exist:

Browser: It allows control of the simulator and shows the simulation results [15].

Editor: It enables the user to create new charts and to modify existing ones [4].

Layouter: The layouter [3] provides several layout algorithms for automatic layout from within the browser and the editor.

Simulator: It performs the necessary calculations for the simulation of the charts. Currently, a simulator for *Esterel Studio* charts is implemented [9].

File Interface: Several import and export filters exist as plug-ins to load and save charts and simulation input data [14]. Currently, it is possible to load and save *Stateflow* charts and *Esterel Studio* charts. ESI and Matlab M files can be read and saved.

KielFrame: This is the main component that provides the program's main window. The main window displays either the browser or the editor. It is possible to switch between both at any time. *KielFrame* also exists to separate all components of the program to keep them independent from each other. Therefore, it encapsulates the interfaces to the simulator, the layouter, and to the file interface. Neither the browser nor the editor calls any function of the other components directly; they call functions of the *KielFrame* instead, which then calls the appropriate functions of the modules.

3.6.2. Communication with *Matlab*

Usually, the user enters *Matlab* commands at *Matlab's* command line or he writes them into a script file. To use the API from within *KIEL*, one could write a routine that generates a *Matlab* script, and then let *Matlab* execute this script. *Matlab* would write its response into a file that the *KIEL* program can read. This way of communicating with *Matlab* produces unnecessary disk accesses. *Java* provides a way to directly communicate with other applications using input and output streams. These streams can be obtained by the methods `getInputStream()` and

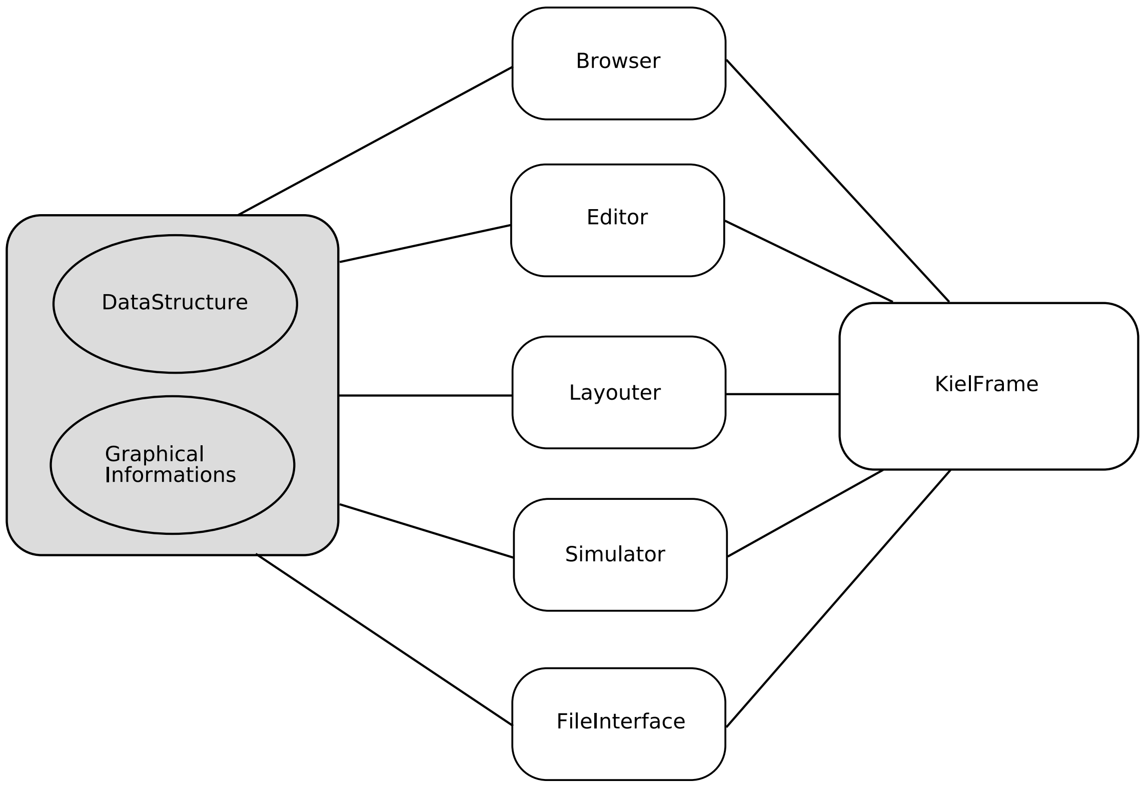


Figure 3.7.: Components of the *KIEL* application

`getOutputStream()` of the class `Process`. An instance of the `Process` class is obtained during the creation of the *Matlab* process with one of the `Runtime.exec` methods.

Example of communication with *Matlab*: Let `states` be a handle to states in one level of hierarchy of a chart that has to be imported. The following messages are exchanged between *KIEL* and *Matlab* to retrieve information about a state.

```
size(states)
```

```
ans =

     2     1

>>
```

```
states(1).Position
```

```
ans =

     312.6471   247.3765    90.0000    60.0000

>>
```

```
states(1).Type
```

```
ans =

     OR

>>
```

```
states(1).IsSubchart
```

```
ans =

     0

>>
```

```
states(1).Id
```

```
ans =

     20

>>
```

```
states(1).Name
```

```
ans =

     off

>>
```

The following 3 sections describe how to use the API to find objects in the *Stateflow* chart, and how to manipulate them as well as how to set the simulation parameters with the API. They provide the details of the communication with *Matlab*. For an introduction to the API have a look at Section 2.3.

3.6.3. Finding *Stateflow* Objects using the API

The `find` method provides a way to find objects in the diagram. Practically, this method searches for all objects that are contained in the object the `find` method belongs to. It is possible to restrict the search depth. Therefore, it is not necessary to traverse the diagram object by object to find what you are searching for. The `find` method needs to know the type of object it has to find and, optionally, one or more property names and their values.
Example: Let `m` denote a variable containing a handle to a model object. Then the following command will return every State object with the name "on" in the model `m`:

```
onState = m.find('-isa','Stateflow.State',
    '-and','Name','On')
```

If `find` finds more than one object, then it will return an array of objects. It is important to note that the zeroth level of containment is always included in the answer of `find`. To limit the search depth, one uses the `-depth` specifier.
Example: Let `sA` denote a variable containing a handle to a state object. Then the following command will return every object that `sA` contains up to a level of containment of 2:

```
objArray = sA.find('-depth',2)
```

Using `find` one can find child objects only. To find the parent object, one has to use the `up` method.

Getting and setting the Properties of Objects

One can get the property of an object just by writing down the object handle variable followed by a "." and the property, or by using the `get` method. Example:

```
d = obj.Description
d = obj.get('Description')
```

One can set the property of an object by using the assignment operator or the `set` method. Example:

```
obj.Description = 'This␣is␣a␣description␣for␣the␣object.'
obj.set('Description','This␣is␣a␣description␣of␣the␣object')
```

3.6.4. Setting the Simulation Parameters using the API

The simulation parameters can be set using the API. The object `Simulink.ConfigSet` contains all settings for the simulation and for code generation. The settings appear in the Model Explorer as the item *Configuration (Active)*. One can get the actual configuration with the function `getActiveConfigSet`. Let `acs` denote the variable to which we want to save the handle to the current configuration object, and let `model` denote the variable containing a handle to the current model. Then the following command will store a handle to the actual configuration object into the variable `acs`:

```
acs = getActiveConfigSet(model)
```

The function `set_param` provides the possibility to change the values of parameters of the configuration object. To choose the discrete solver that is of type fixed-step, one has to enter the following two commands:

```
acs.set_param('SolverType','Fixed-step')
acs.set_param('Solver','FixedStepDiscrete')
```

Finally, we will need a way to change the simulation stop time and the sample time:

```
acs.set_param('StopTime','5')
acs.set_param('FixedStep','1.0')
```

In the example above, we choose a stop time of 5 seconds and a sample time of 1 second.

3.6.5. Manipulating the *Simulink* Model using the API

It is possible to modify the *Simulink* model with the help of the API. Commands exist to add, delete, and link *Simulink* blocks. To add a block to the model, one uses the `add_block('src','dst')` command. `add_block` copies the block with the full path name `'src'` to a new block with the full path name `'dst'`. The *Simulink* library provides an extensive set of blocks. One usually builds a model using these predefined blocks.

The parameters of the new block are identical to those of the original. The `set_param('obj', 'parameter1', value1, 'parameter2', value2, ...)` function can be used to change the parameters of the block specified by `'obj'`. Pairs of parameters and values are given to `set_param` to set the parameters to the new values. To position a block, one changes the `position` parameter to a vector with coordinates according to the following scheme: `[left top right bottom]`. The origin of the coordinate system is the upper left corner.

One can link two blocks together using the `add_line('sys','oport','iport')` command. It adds a line from the output port of one block to the input port of the other block. `'iport'` and `'oport'` which consist of a block name and a block port identifier in the form `block/'port'`. Usually, ports are numbered from top to bottom or from left to right.

3.6.6. Import of *Stateflow* Charts

It is assumed that the user who wants to work with *Stateflow* charts has a *Matlab/Simulink/Stateflow* license. Since the *KIEL* application is not meant to replace *Matlab/Simulink/Stateflow*, the solution of the import problem may require it. One possible solution is to write a parser for *Simulink* model files. Every time the format of these model files changes, the parser has to be adapted. It has shown that the model file format is not documented. It has a great similarity to the API objects, but there are differences. Another possible solution is to let *Matlab* load the model file itself, and then use its API to get information about the chart object by object. This information is used to create an equivalent chart using the *KIEL* data structure. The API is well documented and unlikely to undergo dramatic changes in the future. This solution is therefore easier to accomplish. The second solution has been implemented and is described in detail in the following part. The class *MatlabStateflowGrabber* includes all the program code that is necessary to import a chart from *Matlab*.

The *KIEL* application starts *Matlab* when a *Stateflow* model file (file name extension “mdl”) has to be loaded. It then writes *Matlab* commands to the input stream associated with the *Matlab* process and receives *Matlab's* response through the corresponding output stream. *Matlab's* responses contain all the information that is needed to construct an equivalent chart using the *KIEL* data structure.

Import Strategy

The algorithm is shown in pseudocode in Figure 3.8. Starting at the root, the chart is traversed downwards level by level of containment. This allows the reuse of variables in the *Matlab* workspace and avoids a recursive approach. For each object in the chart an equivalent object is created using the classes of the *KIEL* data structure. Since interlevel transitions are not supported by *KIEL*, it is possible to complete the constrution of one level of containment before the next level is entered. States have to be constructed before transitions are constructed, otherwise the algorithm might try to link a transition to a state which has not been created yet. The transitions in *Stateflow* refer to their source and destination states. While the transitions are generated in *KIEL*, it is necessary to associate the states in *Stateflow* with the states created in *KIEL*. The association is realized with a `HashMap` of the `java.util` package. The states have ids in *Stateflow* that are used as keys for the `HashMap`.

Delimiter lines are used in *KIEL* to separate regions from each other. They do not appear in *Stateflow* charts because regions are drawn as dashed rectangles instead. Therefore, it is necessary to generate them during the process of import. To calculate the coordinates for the delimiter lines, the regions are sorted from left to right and from up to down, so that the upper left region is the first one and the lower right region is the last one. Between two adjacent regions a vertical delimiter line is set if they lie side by side, a horizontal line is set if they lie one upon another.

The positions of graphical objects are absolute in *Stateflow* according to the coordinate system of its editor. In contrast to *Stateflow*, the positions are relative to the positions of their parent objects in *KIEL*. During import, all position coordinates are converted from absolute to relative coordinates.

Stateflow supports the concept of subcharts. A subchart is a state that contains a complete chart. This concept is used to hide complexity. The contents of a subchart is never shown together with its parent. The *KIEL* application does not support this concept. Therefore, the original layout cannot be imported for charts which contain subcharts. One of the automatic layouters is invoked if a chart that has to be imported contains one or more subcharts.

Actions are represented as objects in the *KIEL* data structure. In contrast to this, there is no API object to represent actions. Actions are just stored as string labels in states and transitions. It is necessary to extract the actions from the state and transition labels. For each action that appears in a label within the chart we import, an object is created. However, the contents of the action is not parsed because the simulation is done by *Stateflow*, so we do not need to know what the action consists of, but we need to know which actions exist and of which type they are to be able to represent them correctly in the *KIEL* application. The contents of an action element is stored as an unparsed string in *KIEL*.

The spatial coordinates of transition destination end points are not available through the API. Instead, transition destination end points are specified as o'clock positions in the range of 0 to 12. The spatial coordinates are calculated from the o'clock value and the spatial coordinates of the center of the destination object. The spacial coordinates of transition source end points are available through the API. They are just transformed into coordinates that are relative to the upper left corner of the parent state. In addition to the source end and destination end point each transition in *Stateflow* has a midpoint. These 3 points are used to construct a path which consists of a `MovetoPath` and a `QuadraticCurvetoPath` path element. The `MovetoPath` path element contains the transition source end point as its target point. The `QuadraticCurvetoPath` path element contains the transition source end point and the transition midpoint as its supporting points and the transition destination end point as its target point. A path contains the graphical information which determines the layout of a transition.

```
procedure grabChart
  set chart to ‘‘empty’’
  call getAndCreateGlobalEvents with chart and rootState
  call getAndCreateGlobalVariables with chart and rootState
  call grabOneLevel with rootState and chart returning children
  set parentStates to children

  repeat
    set nextLevelParentStates to ‘‘empty’’

    for each element in parentStates
      set parent to element
      call grabOneLevel with parent and chart returning children
      append children to nextLevelParentStates
    end for

    set parentStates to nextLevelParentStates
  until nextLevelParentStates contains more than zero elements
  return chart
end procedure

procedure grabOneLevel(parent, chart)
  set children to children of parent

  for each element in children
    call getAndCreateStates with element and chart
    call getAndCreateVariables with element and chart
    call getAndCreateEvents with element and chart
    call getAndCreateJunctions with element and chart
    call grabAndCreateTransitions with element and chart
  end for
  call createDelimiterLines with children and chart
  return children
end procedure
```

Figure 3.8.: Algorithm in pseudocode for import of *Stateflow* charts

3.6.7. Export of *Stateflow* Charts

To export a chart it is possible to write a mdl file directly, which is difficult due to the lack of a good and complete documentation of the file format. Therefore, the same API based approach as for the import feature has been chosen for the export feature. In a first step, a new chart that is equivalent to the current chart in the *KIEL* application is generated in *Stateflow* with the use of the API. In a second step, *Matlab* is told to save the new chart to a file. In case the user switches from the *KIEL* editor to the browser, the second step is omitted. The class `MatlabStateflowCreator` implements the *Stateflow* chart export feature.

Export strategy

The algorithm is shown in pseudocode in Figure 3.9. The chart is created level by level of containment in a non-recursive manner. For each level, states and junctions are created before any transitions are created to avoid the situation where a transition is going to be linked with a state or a junction that has not been created so far.

To confer unique names to variables in the *Matlab* workspace, the return value of the `java.lang.object.hashCode()` function is used at the end of every variable name. Variable names of variables that contain handles to states consist of the state's name followed by an underscore and the corresponding hash code of the object that contains the state. Variable names of variables that contain handles to transitions consist of the prefix `trans_` and the hash code of the object that contains the transitions. Variable names of variables that contain handles to junctions or history entities are constructed in the same way except the prefix is `junc_` or `hist_` respectively.

The labels of states are constructed out of the name of the state and out of the actions that are associated with the state. Each type of action has a prefix as described in Section 2.2.2.

Delimiter lines are omitted because they do not exist in *Stateflow*. Regions are called parallel states or AND states in *Stateflow* and are drawn as dashed rectangles. The positions of graphical objects are relative to their parent in *KIEL*, but absolute in *Stateflow*, so they have to be converted during the export.

The source end points and destination end points of transitions cannot be set directly as spatial coordinates, but they have to be specified as o'clock positions. An o'clock position is in the range of 0 to 12. These points are stored as coordinates in *KIEL*, so the appropriate o'clock value is calculated from the point coordinates and the center point coordinates of the source or destination object respectively. Source end points of default transitions form an exception because an o'clock value makes no sense here. The x and y coordinates are set directly for source end points of default transitions. The spatial coordinates of the transition midpoint are taken from the path element that specifies a point between the source and the destination end point. If the path consists of n elements with $n \geq 3$, then the target point of the n/2-th path element is taken as midpoint. If the path consists of 2 path elements only and the second path element is of type `QuadtraticCurvetoPath`, then the second supporting point of this path element is taken as midpoint. If the path consists of 2 path elements and the second element is of type `lineto`, then the midpoint is calculated from the source point and end point coordinates.

3.6.8. Trace of input data

The package `kiel.simulationTrace` contains the classes `TraceData` and `TraceStep`, which implement the trace feature. An instance of the class `TraceData` holds the input trace data. For each trace step it contains an instance of the class `TraceStep`. A `TraceStep` contains a list of objects of type `Signal` for signals which are present, of type `SignalValue` for valued signals which are present, of type `VariableValue` for variables. It is necessary to load the chart before the trace file is loaded because the objects of type `Signal`, `IntegerSignal`, and `Variable`, which are contained in the loaded chart, are needed to construct the `TraceStep` objects. If the trace file is loaded first, then the trace steps will not contain any objects which makes the loaded trace useless.

3.6.9. Simulation of *Stateflow* Charts

The class `StateflowSimulator` in the package `kiel.simulator.matlab` contains the implementation of the simulator for *Stateflow* charts. It implements the interface `Simulator`. The *Stateflow* simulator does not perform the calculations itself. It just collects the input data from the tables of the browser and sets the variables in the *Matlab* workspace accordingly. Next, it sends the simulation command to *Matlab*, collects the simulation results and creates objects of type `MicroStep` according to these results. These `MicroStep` objects are put into a `MacroStep` object, which is the final result of a simulation step. The order of the micro steps does not represent the order of the computations made by *Stateflow*. The list of micro steps is not complete, but it contains all micro steps that are needed to display the result of a simulation step. Micro steps that are needed are those that determine the active states as well as the values of output signals and output variables. Micro steps that determine which transitions have been checked for validity are not needed. This information cannot be retrieved by *Stateflow* and is therefore unavailable. That is why the micro step simulation mode cannot be implemented for *Stateflow* charts as long as *Stateflow* is used to perform the simulation.

`MicroStep` is an interface that is implemented by the following classes: `Change-Value`, `SignalStatus`, `StateStatus`, and `TransitionStatus`. Figure 3.10 shows the corresponding class diagram. Objects of type `TransitionStatus` are not used by the simulator because the kind of information represented by them is not available. Objects of type `ChangeValue` contain new values for variables and for valued signals. They are used to report changes of values of variables and of valued signals. Direct subclasses are: `SignalValue` and `VariableValue`. The class `SignalStatus` has the

```
procedure createChart
  call createGlobalEvents
  call createGlobalVariables
  call createOneLevel with rootState returning children
  set parentStates to children

  repeat
    set nextLevelParentStates to ‘‘empty’’

    for each element in parentStates
      call createOneLevel with element returning children
      append children to nextLevelParentStates
    end for

    set parentStates to nextLevelParentStates
  until nextLevelParentStates contains no more than zero elements
end procedure

procedure createOneLevel(parent)
  set children to children of parent

  for each element in children
    call createStates with element
    call createJuncions with element
    call createTransitions with element
    call createLocalEvents with element
    call createLocalVariables with element
  end for

  return children
end procedure
```

Figure 3.9.: Algorithm in pseudocode for export of *Stateflow* charts

following direct subclasses: `SignalAbsent`, `SignalPresent`, and `SignalUnknown`. `SignalAbsent` represents the absence of the signal it contains, `SignalPresent` represents the presence of the signal it contains. The status of a signal contained in a `SignalUnknown` object is unknown. An object of type `StateActivated` in the list of micro steps means the state it contains is entered. It contains a list of all states that are currently active. An object of type `StateDeactivated` means the state it contains is leaved. The other types of micro steps that exist are not used in the *Stateflow* simulator.

Before a chart can be simulated, the simulink environment has to be prepared to allow input from and output to the *Matlab* workspace. The browser calls the method `setStateChart(StateChart)` after a chart has been loaded or the user has switched from the editor to the browser. This method deletes all simulink blocks except for the chart block, adds a `Form Workspace` block for each input variable and one for all input signals that is connected to the trigger port of the chart. For each state, the `HasOuputData` option is set to true. This results in an output signal for each state whose value is 1 for every active state and 0 for every non-active state. All output signals are connected to a `Mux` block whose output is connected to a `To Workspace` block named `simout`. For each output variable, the option `SaveToWorkspace` is set to true. Their output ports remain unconnected. Section 3.6.5 explains how to manipulate the *Simulink* environment using the API. The solver type is set to `Fixed-step`, the solver to `FixedStepDiscrete`. The sample time is set to 1 second. Section 3.6.4 explains how to set the simulation parameters using the API.

If the user clicks onto the `MacroStep` button in the simulation toolbar of the browser and the browser is in the signal simulation mode, then the browser calls the method `emit(Event, int)` for each input signal of the chart. The integer parameter is the value of the signal. If the browser is in the event simulation mode while the user clicks onto `MacroStep`, then it calls the method `emit(Event)` instead. The browser calls the method `setVariable(Variable, String)` for each input variable. The second parameter is the string representation of the value of the variable. After that, it calls the method `nextStep()` which performs one simulation step and returns a `MacroStep` object that includes all simulation results. All states that are included in the list of active states in the last `StateActivated` object are marked as being active by the browser. For each `VariableValue` object, the browser changes the value of the corresponding variable to the new value contained in that object.

The method `nextStep()` constructs a matrix which contains the values of the input signals for each simulation step. Let m denote the total number of simulation steps and n the total number of input ports. Then the matrix has m rows and $(n + 1)$ columns. Column 1 row j contains the number j of the step, the entry in column i $(1 < i \leq n)$, row j, contains the value at step j of the input signal whose port number is i. There is a row for each simulation step. The matrix is stored in a variable defined in the workspace. The name of the variable is equal to the name of the `From Workspace` block that is connected to the trigger port of the chart. Due to this name equality *Simulink* attaches the contents of the variable to this `From`

`Workspace` block. For each input variable a variable with equal name plus the postfix "VarMat" is defined in the workspace. The corresponding `From Workspace` blocks also have these names. Due to this name equality *Simulink* attaches the contents of these variables to the coresponding `From Workspace` blocks. Such a variable in the workspace contains a matrix with as many rows as there are simulation steps to simulate and 2 columns. The first column contains the step numbers and the second column contains the values of the variable at the coresponding step. After all these variables that include the input data for the simulation are defined, the method `nextStep` sets the parameter `StopTime` to the current step number (Each step number equals to 1 second.) and invokes the function `sim`, which belongs to the API. *Simulink* simulates the chart and stores the simulation results in variables that are defined in the workspace. For each output variable, *Simulink* creates a variable in the workspace whose name is equal to the name of the variable in the chart. The values of these variables are those determined at the end of the simulation. The variable `simout` contains the values of all output signals at each step as a matrix. This includes the output signals that report whether a particular state is active at a particular simulation step. The simulator examines the last two rows of this matrix. (The last row corresponds to the last simulation step and therefore contains the final simulation results.) It creates a `StateActivated` object for each state that is active in the last step and has been inactive in the previous step, and it creates a `StateDeactive` object for each step that is inactive in the last step and has been inactive in the previous step. Every active state that does not contain any states is added to the list of currently active states. This list is used to create a `Configuration` object. For each output signal that does not belong to a state, the simulator creates a `SignalPresent` object if the signal value switches between 1 and 0 (event is present) and a `SignalAbsent` object if the signal switches between 1 and 0 (event is not present). The simulator creates a `VariableValue` object for every output variable and local variable whose value has changed. All created micro steps are put into a `MacroStep` object which is returned as result by the method `nextStep`.

3.6.10. Adaption of the Editor

The type of the chart is stored in the field `modelSource` of the class `StateChart`. The editor calls the method `getModelSource()` to check whether the current chart is designed for *Esterel Studio* or for *Stateflow*. The toolbar is constructed by the method `updateToolBarAndMenuBar()` of the class `Editor` in dependence of the type of the current chart. The items the toolbar consists of are instances of the classes found in the package `kiel.editor.controller`. The class `EditorModeAddJunction` has been added to the package to support the creation of junctions. The graphical representation of a junction is determined by the class `JunctionRenderer`, which has been added to the package `kiel.editor.graph`. The mouse cursor that is shown while a junction is placed on the canvas is created using the image `JunctionAndCursor.gif`, which resides in the package `kiel.editor.resources`. The method `getMenuItems-`

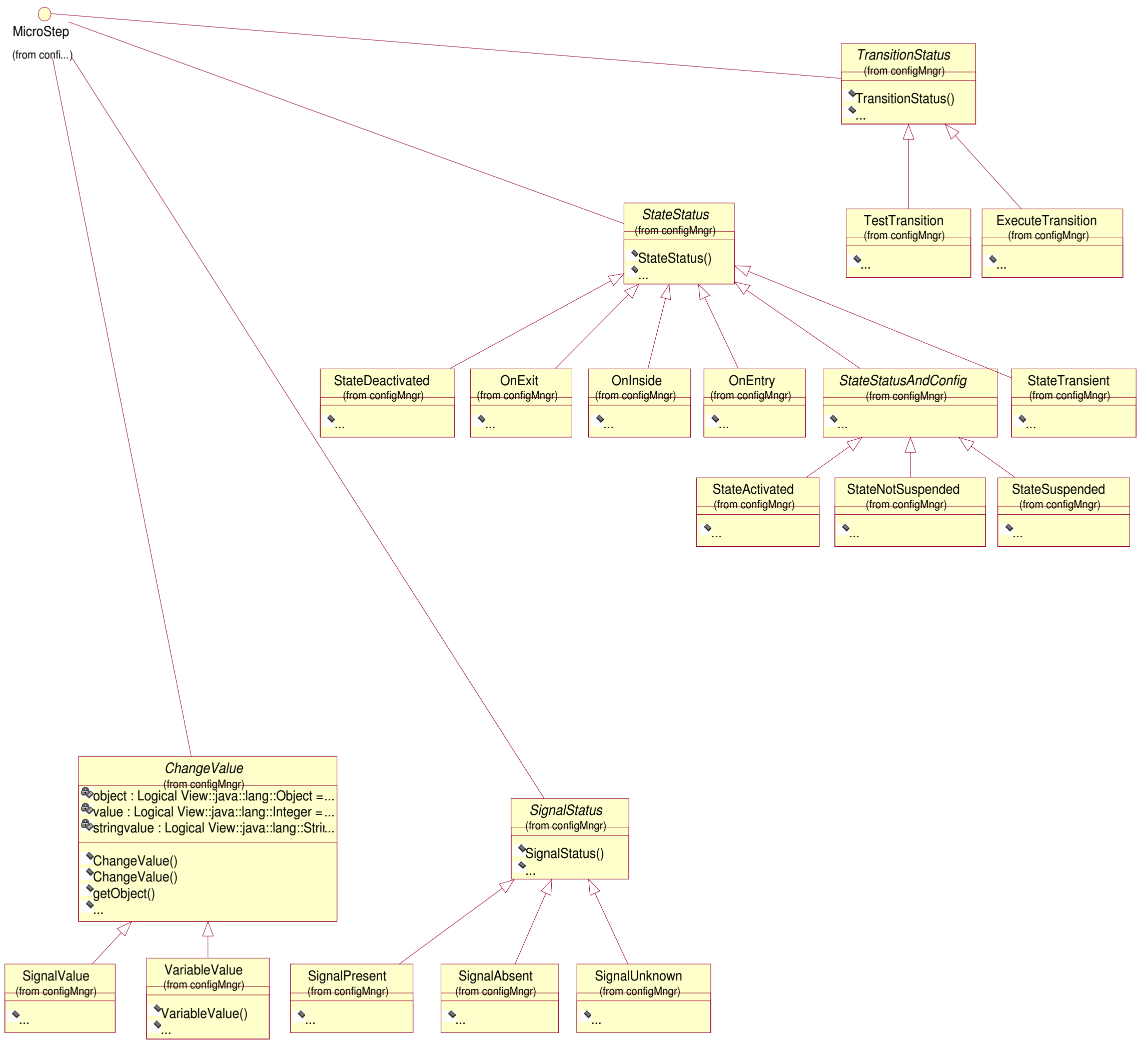


Figure 3.10.: The interface MicroStep and the classes that implement it

`ForSelected()` of the class `MyJGraph` determines the menu items that appear in the context menu. In *Stateflow* mode, the items to change the type of a transition and the item that converts a state into a final state are omitted.

The reaction of the editor to user input is implemented by the classes in the package `kiel.editor.controller`. To implement new dialogs for editing variables and events without changing the dialogs that already exist for *Esterel Studio* charts, the following new controller classes have been added: `ActionEditStateChartEvents-ForStateflow`, `ActionEditStateChartVariablesForStateflow`, `ActionEditState-EventsForStateflow`, `ActionEditStateVariablesForStateflow`. As dialog window a `javax.swing.JDialog` is used. Variables and events are displayed in a table. This table is an instance of `javax.swing.JTable`. The classes that handle the user input of transition labels and actions in states cannot be used for *Stateflow* charts because they parse the entered strings with the help of the compound label parser that has been developed to support *Esterel Studio* charts. The following classes have been added to support unparsed user input of lables and actions: `ActionEditState-EntryActionsForStateflow`, `ActionEditStateDoActionsForStateflow`, `ActionEdit-StateExitActionsForStateflow`, and `ActionEditStateBindActionsForStateflow`.

4. Concluding Remarks

This chapter summerizes the results of this work and mentions possible future enhancements. It includes the results of a performance test.

4.1. Results

The task was to add the ability to load, save, create, edit, and simulate *Stateflow* charts to the *KIEL* application. This required changes in the file interface, the browser, and the editor. These features were implemented with the help of the *Stateflow* application, which runs concurrently to *KIEL* and performs the core of the task. The communication between *KIEL* and *Matlab/Simulink/Stateflow* has been implemented using input and output streams. The commands that are exchanged are those defined by the *Simulink* and *Stateflow* API. The toolbar and the menu items of the editor as well as the dialogs for editing variables and events have been adapted to support *Stateflow* charts.

All goals mentioned in Section 3.1 have been achieved. The new *KIEL* application is well suited for the creation and the simulation of *Stateflow* charts. However, it lacks support for *Simulink* and for code generation. Therefore, it cannot serve as a replacement for *Matlab/Simulink/Stateflow*, but it is a good extension for it.

The execution of and the communication with *Matlab* consumes calculation time. The performance of *KIEL* with respect to loading, saving, and simulating *Stateflow* charts has been tested with charts of different complexity. The first loading or saving of a *Stateflow* chart requires the start of *Matlab*. If *Matlab* has already been started by *KIEL*, then the process of loading or saving a chart consumes less time. The first simulation step requires the compilation of the chart by *Stateflow*, which consumes additional time. All further simulation steps consume less time. Table 4.1 shows the results of time measurements for loading and simulation of four charts with different complexity, which are shown in Figure 4.1 to Figure 4.3. The measurements have been made on an AMD Athlon XP 3000+ processor with 1 GB RAM.

4.2. Future Prospects

The simulator of *KIEL* deletes all blocks in the *Simulink* diagram except for the chart block and constructs its own *Simulink* diagram. Therefore, it is not possible to simulate an existing chart within the *Simulink* diagram it has been developed for. A future version of the simulator might preserve the existing *Simulink* diagram.

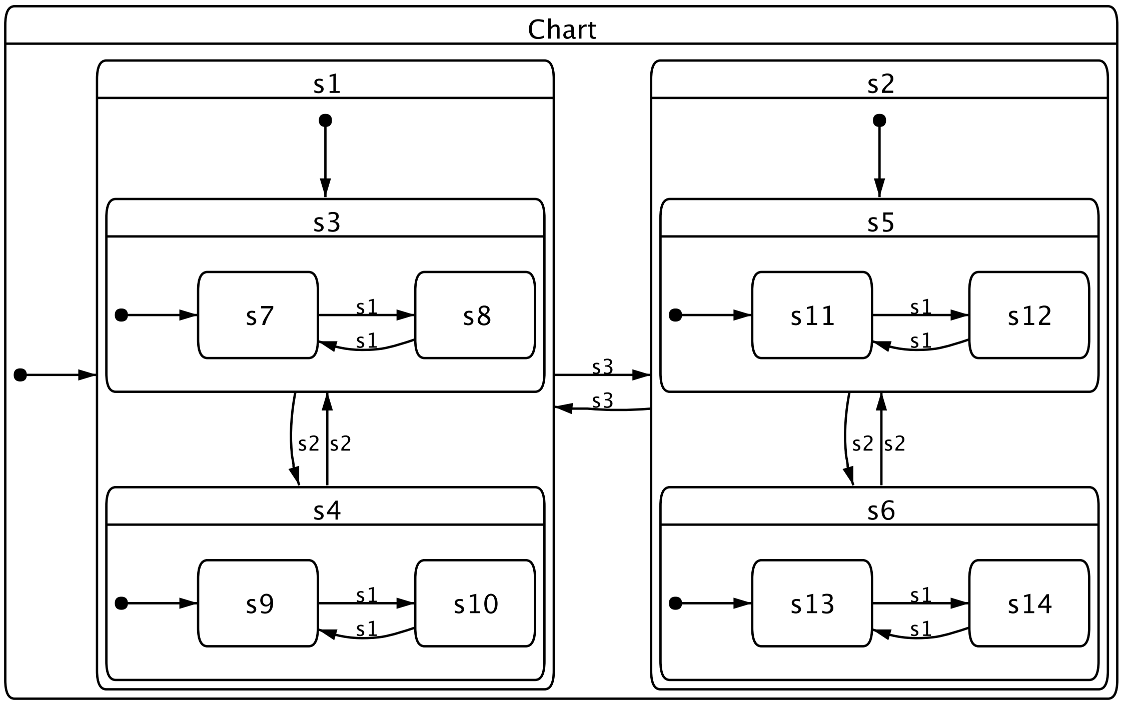


Figure 4.1.: The bintree3 Chart

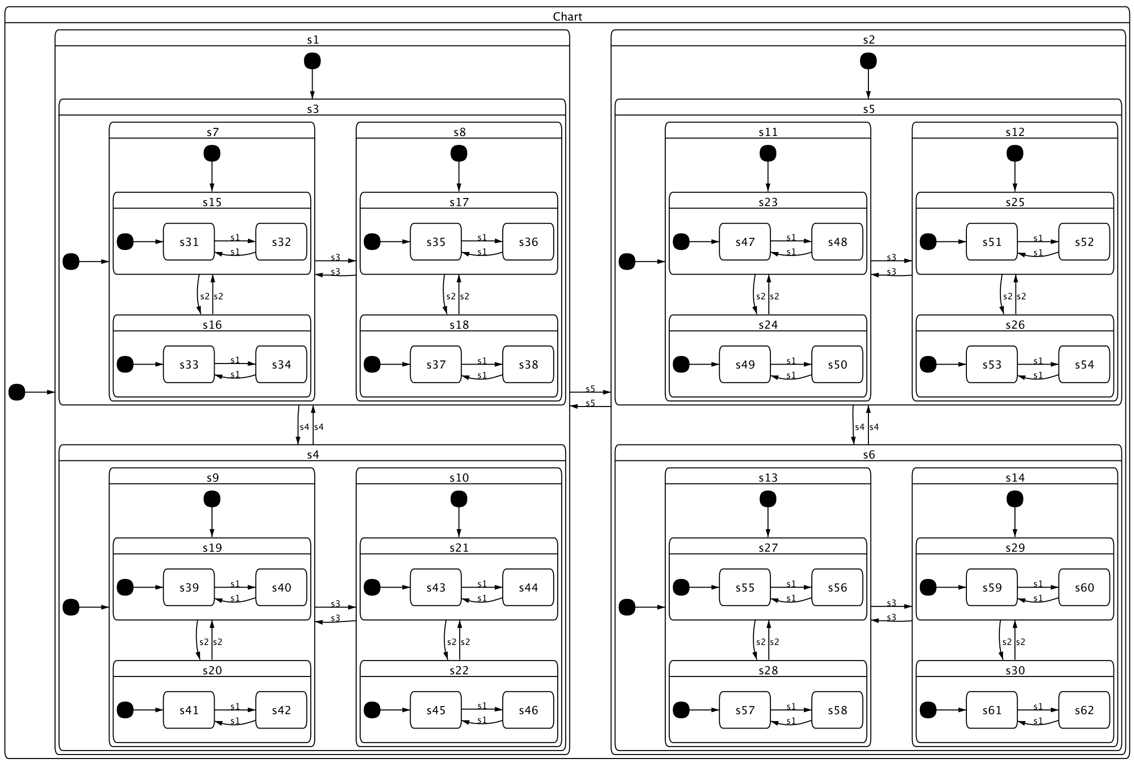


Figure 4.2.: The bintree5 Chart

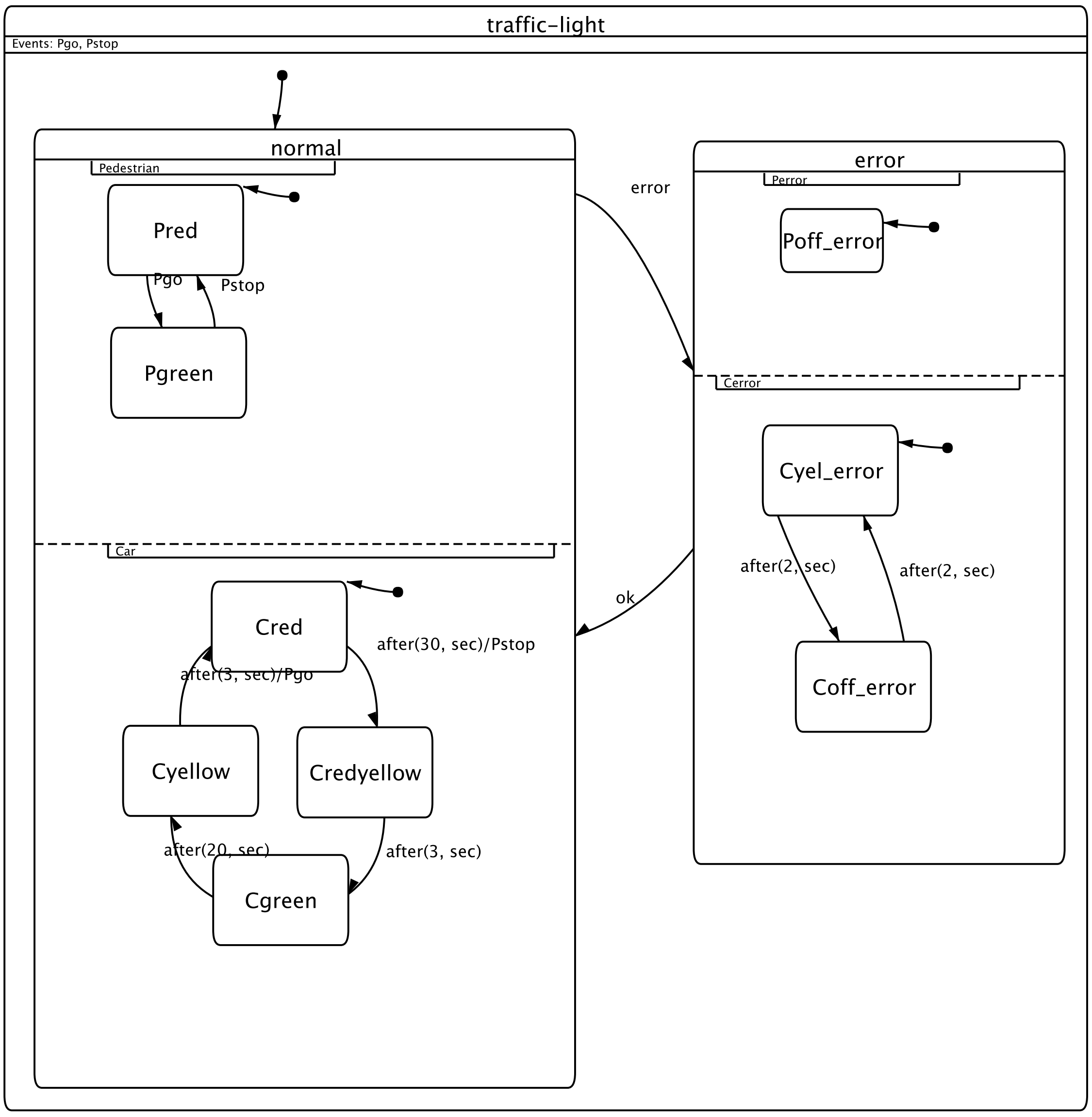


Figure 4.3.: The traffic light chart

Chart	States	Transitions	loading Time	min Simulation Step Time	max Simulation Step Time
bintree3	21	35	934 ms	221 ms	798 ms
bintree5	93	155	1764 ms	341 ms	2162 ms
bintree7	381	635	3993 ms	889 ms	11176 ms
traffic-light	15	25	2236 ms	249 ms	645 ms

Table 4.1.: Performace test results

This approach does not provide the same level of control over the input to the chart because some input data is usually provided by other *Simulink* blocks. It is necessary to tap the input signals (lines) to be able to display the input data at each simulation step. To tap these signals, the simulator needs to analyse the *Simulink* diagram to determine which blocks provide input to the chart as well as the coordinates of these blocks. To get the output data, one can either tap the output signals in the same way as the input signals or one can use the "save to workspace" feature of *Simulink*. See Section "Exporting Output Data to the *Matlab* workspace" and "Logging Signals" in the *Simulink* documentation [5] for further details.

The *Stateflow* diagram elements Truth Table, Graphical Function, Box, embedded Matlab Function, and Note are currently not supported. Therefore, charts that make use of these elements can neither be imported into nor be created with *KIEL*. To add support for these diagram elements, it is necessary to add appropriate classes to the *KIEL* data structure and to extend the browser, the editor, and the simulator. A graphical representation for these diagram elements has to be implemented in the browser and the editor.

Interlevel transitions are currently not supported. The import and export algorithms for *Stateflow* charts used here cannot handle such transitions. The layouter module cannot handle such transitions either because the layout algorithms used there have been designed for graphs which do not support the concept of hierarchy.

5. Bibliography

[1] Esterel Technologies. Company homepage. http://www.esterel-technologies.com.

[2] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[3] Tobias Kloss. Automatisches Layout von Statecharts unter Verwendung von GraphViz. Diplomarbeit, Christian-Albrechts-Universität zu Kiel, Institut für Informatik und Praktische Mathematik, May 2005.

[4] Florian Lüpke. Implementierung eines Statechart-Editors mit layoutbasierten Bearbeitungshilfen. Diplomarbeit, Christian-Albrechts-Universität zu Kiel, Institut für Informatik und Praktische Mathematik, June 2005.

[5] Mathworks Inc. *Simulink – Simulation and Model-Based Design.* Mathworks Inc., 2005. URL http://www.mathworks.com/access/helpdesk/help/pdf_doc/simulink/sl_using.pdf.

[6] Mathworks Inc. Stateflow Application Programming Interface, 2005. http://www.mathworks.com/access/helpdesk/help/toolbox/stateflow/.

[7] The MathWorks, Inc. *Stateflow and Stateflow Coder User's Guide, Version 6.* Natick, MA, September 2005. URL http://www.mathworks.com/access/helpdesk/help/pdf_doc/stateflow/sf_ug.pdf.

[8] Object Management Group. OMG Unified Modeling Language Specification, Version 1.5, March 2003. http://www.omg.org/cgi-bin/doc?formal/03-03-01.

[9] André Ohlhoff. Simulating the Behavior of Synccharts. Studienarbeit, Christian-Albrechts-Universität zu Kiel, Institut für Informatik und Praktische Mathematik, November 2004.

[10] Jan Täubrich. Untersuchung der nicht-interaktiven Simulation von Stateflow-Statecharts. Praktikumsbericht, Christian-Albrechts-Universität zu Kiel, Institut für Informatik und Praktische Mathematik, April 2005.

[11] The KIEL Project. Project Homepage, 2004. URL http://www.informatik.uni-kiel.de/~rt-kiel/. Kiel Integrated Environment for Layout.

[12] The KIEL Project. Project API Documentation, 2004. URL `http://www.informatik.uni-kiel.de/~rt-kiel/kiel/kiel/doc/`. Kiel Integrated Environment for Layout.

[13] The Mathworks, Inc. Company homepage. `http://www.mathworks.com`.

[14] Mirko Wischer. Ein FileInterface für das KIEL Projekt. Praktikumsbericht, Christian-Albrechts-Universität zu Kiel, Institut für Informatik und Praktische Mathematik, 2005.

[15] Mirko Wischer. Ein Browser für die Visualisierung dynamischer Sichten von Statecharts. Diplomarbeit, Christian-Albrechts-Universität zu Kiel, Institut für Informatik und Praktische Mathematik, 2005.

A. Settings

This chapter describes the available settings for the *Stateflow* simulator and for the file interface plug-in for ".mdl" files.

MatlabProcessInfo.matlabExecutableFileName

This parameter specifies the file name of the *Matlab* executable file. It may also specify the complete path to the executable file.

matlabPrompt

This parameter specifies the matlab prompt. The standard setting is "» ", but this may differ from version to version, especially for student versions.

matlabErrorIndicator

It specifies the string that *Matlab* outputs in error messages. Error messages are recognized by this string. The standard setting is "???".

MatlabStateflowGrabber.chartScale

This factor is used to scale the chart during import. A factor greater than 1 may help to solve the problem of labels that do not fit into their states because of the font size. The standard setting is 1.

MatlabProcessInfo.nodisplay

This parameter specifies whether *Matlab*, which is started by *KIEL*, will display any windows. Possible values are "true" and "false". The standard setting is "true".

StateflowSimulator.inputVariableNamePostfix

This parameter specifies the string that is attached to "From Workspace" blocks for input variables. The standard setting is "VarMat".

B. Java Code

This chapter introduces all implemented classes. It includes the complete program code of all extensions that have been made to the *KIEL* application.

B.1. Description

B.1.1. Package `kiel.fileInterface.matlab`

This package includes the plug-in for "mdl" files.

`MatlabStateflowFileFilter`: This class is used in the file dialog to filter files according to their file name extension. Only files with the file name extension ".mdl" are returned by this filter.

`MatlabStateflow`: This class contains the code that is used by the file interface to load and save ".mdl" files. It starts *Matlab* and uses the classes `MatlabStateflowGrabber` and `MatlabStateflowCreator` to do the real work of import and export.

B.1.2. Package `kiel.fileInterface.M`

This package includes the plug-in for "m" files. This plug-in supports only files that include simulation input data. It is used for the input trace feature.

`MFileFilter`: This class is used in the file dialog to filter files according to their file name extension. Only files with the file name extension ".m" are returned by this filter.

`MFileFormat`: This class contains the code that is used by the file interface to load and save ".m" files. It creates a vector variable for each input signal and each input variable with the corresponding values for each simulation step. It cannot handle all features of this file format. It can be used in conjunction with the input trace feature only.

B.1.3. Package `kiel.fileInterface.ESI`

This package includes the plug-in for ".esi" files. This file format is used in *Esterel Studio* for the input trace feature.

`ESIFileFilter`: This class is used in the file dialog to filter files according to their file name extension. Only files with the file name extension ".esi" are returned by this filter.

`ESIFileFormat`: This class contains the code that is used by the file interface to load and save ".esi" files.

B.1.4. Package `kiel.util`

This package contains utility classes, which can be used by any other class of the *KIEL* application. Some of these classes are of special interest for this work.

`MatlabStateflowGrabber`: It uses the *Stateflow* API to collect all needed information about a chart in *Stateflow* and constructs an equivalent chart using the *KIEL* data structure.

`MatlabStateflowCreator`: It uses the *Stateflow* API to construct a chart in *Stateflow* that is equivalent to the chart currently loaded in *KIEL*.

`MatlabProcessInfo`: This class holds a reference to the *Matlab* process and its input and output stream. It is responsible for the start and termination of *Matlab*.

`MatlabStateflowProperties`: It is used to load and to store settings of components that are related to *Stateflow*.

B.1.5. Package `kiel.simulator.matlab`

This class contains the simulator for *Stateflow* charts.

`StateflowSimulator`: This class implements the simulator for *Stateflow* charts. The simulator uses *Matlab/Simulink/Stateflow* to do the simulation.

`SimulatorException`: This exception is thrown by the stateflow simulator in case an error occurs with the input or output streams or *Matlab* reports an error.

B.2. kiel.fileInterface.matlab

B.2.1. MatlabStateflowFileFilter

```java
//$Id: MatlabStateflowFileFilter.java,v 1.1 2005/06/03 12:41:24 apo Exp $
package kiel.fileInterface.matlab;

import java.io.File;

import javax.swing.filechooser.FileFilter;

/**
 * @author Adrian Posor
 * @version $Revision: 1.1 $ last modified $Date: 2005/06/03 12:41:24 $
 */
public class MatlabStateflowFileFilter extends FileFilter {
        /**An instance counter.*/
        private static MatlabStateflowFileFilter instance = null;

        /**Ensures that only one instance exists.*/
        public static MatlabStateflowFileFilter getInstance(){
                if(instance == null){
                        instance = new MatlabStateflowFileFilter();
                }
                return instance;
        }

        /* (non-Javadoc)
         * @see javax.swing.filechooser.FileFilter#accept(java.io.File)
         */
        public boolean accept(File f) {
                if(f.isDirectory()){
                        return true;
                }
                return (
                                f.canRead() && f.isFile()
                                && f.getPath().toLowerCase().endsWith(".mdl"));
        }

        /* (non-Javadoc)
         * @see javax.swing.filechooser.FileFilter#getDescription()
         */
        public String getDescription() {
                return "Matlab/Simulink/Stateflow";
        }
}
```

B.2.2. MatlabStateflow

```java
//$Id: MatlabStateflow.java,v 1.10 2005/12/15 16:13:48 apo Exp $
package kiel.fileInterface.matlab;

import java.io.File;
import java.io.IOException;
import java.io.PrintWriter;

import javax.swing.filechooser.FileFilter;

import kiel.dataStructure.StateChart;
import kiel.fileInterface.FileInterface;
import kiel.fileInterface.FileInterfaceException;
import kiel.graphicalInformations.View;
import kiel.util.Benchmark;
import kiel.util.LogFile;
import kiel.util.MatlabStateflowGrabber;
import kiel.util.MatlabStateflowCreator;
import kiel.util.MatlabProcessInfo;
import kiel.util.MatlabStateflowGrabberException;
import kiel.util.MatlabStateflowProperties;

/**
 * @author <a href="mailto:apo@informatik.uni-kiel.de">Adrian Posor</a>
 * @version $Revision: 1.10 $ last modified $Date: 2005/12/15 16:13:48 $
 */
public class MatlabStateflow extends FileInterface {
    /**
     * The Matlab prompt, usually '>> '.
     */
    private static String matlabPrompt;

    /**
     * The string that indicates an error in Matlab, usually '???'.
     */
    private static String matlabErrorIndicator;

    /**
     * The file the log is written into.
     */
    private LogFile log = null;

    /**
     * The last answer of Matlab.
     */
    private String input = "";

    /**
     * Size of the input buffer (for receiving Matlab's answer).
     */
    private static final int BUFF_SIZE = 20;

    /**
     * Benchmark for loading charts.
     */
    private Benchmark loadBench;

    /**
     * Benchmark for saving charts.
     */
    private Benchmark saveBench;

    /**
     * Benchmark for loading charts.
     */
    private Benchmark startMatlabBench;

    /**
     * Creates an instances of this class. Initializes the fields.
     */
    public MatlabStateflow() {
        log = new LogFile(new PrintWriter(System.out));
        log.setLogLevel(0);
        log.enableLog();
        log.setModuleName("MatlabStateflow");
        matlabPrompt = MatlabStateflowProperties.getMatlabPromptString();
        matlabErrorIndicator =
            MatlabStateflowProperties.getMatlabErrorIndicatorString();
        loadBench = new Benchmark();
        loadBench.setModuleName("FileInterface:␣Load");
        loadBench.setPreMessage("Messuring␣loading␣step␣time");
        saveBench = new Benchmark();
        saveBench.setModuleName("FileInterface:␣Save");
        saveBench.setPreMessage("Messuring␣saving␣step␣time");
        startMatlabBench = new Benchmark();

    }

    /**
     * Receives output from Matlab. This method uses a buffered reader
     * that is connected to Matlab's standard output. The stream is read
     * until a Matlab prompt is encountered. Matlab's answer is stored
     * in the global variable <code>input</code>. If an error is indicated
     * by Matlab, then an exception is thrown.
     * @throws IOException if an error occurs in the input or output stream.
     * @throws FileInterfaceException if an error is encountered
     * by Matlab.
     */
    private void checkMatlabResponse() throws IOException,
        FileInterfaceException {
        char[] inputBuf = new char[BUFF_SIZE];
        int numRead = 0;
        input = "";

        while (!input.endsWith(matlabPrompt)) {
            numRead = MatlabProcessInfo.fromMatlabStream.read(
                    inputBuf, 0, inputBuf.length);
            if (numRead > 0) {
                input += new String(inputBuf, 0, numRead);
            }
        }

        if (input.indexOf(matlabErrorIndicator) != -1) {
            throw new FileInterfaceException(input);
        }
    }

    /**
     * Reads the specified file and returns the chart constructed with
     * the KIEL datastructure.
     * @param f - the file to be read.
     * @return the statechart that is contained in the file.
     * @throws FileInterfaceException if an error is encountered
     * by Matlab.
     */
    public final StateChart readStateChartDocument(final File f)
            throws FileInterfaceException {
        loadBench.printOutputToFile(new File("LoadBench" + f.getName() + ".txt")
     );
        loadBench.start();
        try {

            if (MatlabProcessInfo.matlabProcess == null) {
                startMatlabBench.start();
                log.log(0, "Starting␣Matlab...");
                input = MatlabProcessInfo.startMatlab();
                startMatlabBench.stop();
                startMatlabBench.finishBenchmark();
                log.log(0, input);
            }


            MatlabProcessInfo.toMatlabStream.write("sfexit");
            MatlabProcessInfo.toMatlabStream.newLine();
            MatlabProcessInfo.toMatlabStream.flush();
            MatlabProcessInfo.fromMatlabStream.skip(3);
            log.log(0, "Telling␣Matlab␣to␣load␣" + f.getAbsolutePath());
            MatlabProcessInfo.toMatlabStream.write(
                    "open_system('" + f.getAbsolutePath() + "')");
            MatlabProcessInfo.toMatlabStream.newLine();
            MatlabProcessInfo.toMatlabStream.flush();

            checkMatlabResponse();

            log.log(0, input);
            log.log(0, "Done");
            MatlabStateflowGrabber.currentFileName = f.getName();
            loadBench.stop();
            loadBench.finishBenchmark();
            return MatlabStateflowGrabber.grabStateflow();
        } catch (IOException e) {
            MatlabStateflowGrabber.closeLogFile();
            throw new FileInterfaceException("An␣error␣occured␣while"
                    + "starting␣Matlab␣or␣communicating␣with␣it:␣" + e);
        } catch (MatlabStateflowGrabberException me) {
            MatlabStateflowGrabber.closeLogFile();
            throw new FileInterfaceException("MatlabStateflowGrabber:␣"
                    + "Internal␣Error:␣Sent␣wrong␣command␣to␣Matlab.␣" + me);
        }

    }

    /**
     * Returns the view that has been constructed while reading the file
     * with <code>readStateChartDocument</code>.
     * @return the previously constructed view. Null if the chart contains
     * subcharts.
     * @throws FileInterfaceException if an error is encountered
     * by Matlab.
     */
    public final View getReadView() throws FileInterfaceException {
        return MatlabStateflowGrabber.getView();
    }

    /**
     * Writes the chart specified by the parameters <code>s</code> and <code>
     * v</code> into the file specified by <code>f</code>.
     * @param s - the chart to be saved. Only structural information is given
     * here.
     * @param v - the corresponding view. Only graphical information is given
     * here.
     * @param f - the file name of the file the chart has to be saved into.
     * @return the file name of the file that has been stored. May differ
     * in the file extension from the file name specified by the parameter.
     *
     */
    public final File writeStateChartDocument(final StateChart s,
                                              final View v,
                                              final File f) {
        loadBench.printOutputToFile(new File("SaveBench" + f.getName() + ".txt")
     );
        loadBench.start();
        try {
            MatlabStateflowCreator.createChart(s, v);
            MatlabProcessInfo.toMatlabStream.write("sfsave('untitled',␣'"
                + f.getAbsolutePath() + "')");
            MatlabProcessInfo.toMatlabStream.newLine();
            MatlabProcessInfo.toMatlabStream.flush();
            checkMatlabResponse();
        } catch (Exception ex) {
            MatlabStateflowCreator.closeLogFile();
            log.log(0, "An␣error␣occured␣while␣saving␣the␣chart:␣"
                    + ex.getMessage());
        }
        if (!f.getAbsolutePath().endsWith(".mdl")) {
            return new File(f.getAbsolutePath() + ".mdl");
        }
        saveBench.stop();
        saveBench.finishBenchmark();
        return f;
    }

    /**
     * Returns an instance of the file filter for 'mdl' files.
     * @return the file filter.
     */
    public final FileFilter getStateChartFileFilter() {
        return MatlabStateflowFileFilter.getInstance();
    }

    /**
     * Determines whether this plug-in can write files.
     * @return always <code>true</code>.
     */
    public final boolean canWrite() {
        return true;
    }

    /**
     * Determines whether this plug-in can read files.
     * @return always <code>true</code>.
     */
    public final boolean canRead() {
        return true;
    }

    /**
     * Returns the name of this plug-in.
     * @return alsways returns "Matlab/Stateflow Plug-in".
     */
    public final String getName() {
        return "Matlab/Stateflow␣Plug-in";
    }
}
```

B.3. kiel.fileInterface.M

B.3.1. MFileFilter

```java
//$Id:
package kiel.fileInterface.M;

import java.io.File;

import javax.swing.filechooser.FileFilter;


/**
 * @author apo
 * @version $Revision: 1.1 $ last modified $Date: 2005/08/26 09:23:02 $
 */
public class MFileFilter extends FileFilter {
     /**An instance counter.*/
    private static MFileFilter instance = null;

    /**Ensures that only one instance exists.*/
    public static MFileFilter getInstance(){
        if(instance == null){
            instance = new MFileFilter();
        }
        return instance;
    }


    public boolean accept(File f) {
        if(f.isDirectory()){
            return true;
        }
        return (
                f.canRead() && f.isFile()
                && f.getPath().toLowerCase().endsWith(".m"));
    }


    public String getDescription() {
        return new String("Matlab␣Trace␣File");
    }

}
```

B.3.2. MFileFormat

```java
//$Id:
package kiel.fileInterface.M;

import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.LineNumberReader;
import java.util.ArrayList;
import java.util.Iterator;

import javax.swing.filechooser.FileFilter;

import kiel.configMngr.SignalValue;
import kiel.configMngr.VariableValue;
import kiel.dataStructure.Variable;
import kiel.dataStructure.eventexp.Event;
import kiel.dataStructure.eventexp.IntegerSignal;
import kiel.dataStructure.eventexp.Signal;
import kiel.fileInterface.FileInterfaceException;
import kiel.fileInterface.TraceFileInterface;
import kiel.simulationTrace.TraceData;
import kiel.simulationTrace.TraceStep;

/**
 * @author apo
 * @version $Revision: 1.2 $ last modified $Date: 2005/12/15 13:31:12 $
 */
public final class MFileFormat extends TraceFileInterface {
    /**
     * Input variables and input signals.
     */
    private ArrayList inputs;

    /**
     * The string that is currently processed.
     */
    private String currentLine;

    /**
     * The object that is currently added to the trace.
     */
    private Object inputObject;

    /**
     * Reads the file specified by the parameter <code>f</code> and creates
     * a <code>TraceData</code> object according to the contents of the file.
     * @param f - the file to be read.
     * @return the trace that is stored in the file.
     * @throws FileInterfaceException if an error occurs while opening the
     * file.
     */
    public TraceData readTraceFile(final File f)
        throws FileInterfaceException {
        TraceData trace = new TraceData();
        TraceStep step;
        try {
            FileReader datareader = new FileReader(f);
            LineNumberReader datainput = new LineNumberReader(datareader);
            currentLine = datainput.readLine();
            String[] tokenizedLine;
            int stepNumber;

            while (currentLine != null) {
                while ((!currentLine.trim().equals(""))
                        && (!currentLine.startsWith("%"))) {
                    searchForInputObject();
                    stepNumber = 0;
                    if (inputObject != null) {
                        tokenizedLine = currentLine.substring(
                                currentLine.indexOf('=') + 1).trim()
                                .split("\\s+|;\\s*|\\[|\\]");
                        if (inputObject.getClass().equals(
                                IntegerSignal.class)) {
                            Integer recentValue = new Integer(0);
                            trace.seek(-1);
                            for (int i = 1; i < tokenizedLine.length; i += 2) {
                                if (stepNumber == Integer.parseInt(
                                        tokenizedLine[i])) {
                                    if (trace.hasNextStep()) {
                                        step = trace.getNextStep();
                                    } else {
                                        step = new TraceStep();
                                        trace.addStep(step);
                                    }
                                    recentValue = new Integer(
                                            tokenizedLine[i + 1]);
                                    step.add(new SignalValue(
                                            (IntegerSignal) inputObject,
                                            recentValue));
                                    stepNumber++;
                                } else {
                                    int nextStepNumber = Integer.parseInt(
                                            tokenizedLine[i]);
                                    for (int j = stepNumber;
                                        j < nextStepNumber; j++) {
                                        if (trace.hasNextStep()) {
                                            step = trace.getNextStep();
                                        } else {
                                            step = new TraceStep();
                                            trace.addStep(step);
                                        }
                                        step.add(new SignalValue(
                                                (IntegerSignal) inputObject,
                                                recentValue));
                                    }
                                    if (trace.hasNextStep()) {
                                        step = trace.getNextStep();
                                    } else {
                                        step = new TraceStep();
                                        trace.addStep(step);
                                    }
                        recentValue = new Integer(
                                tokenizedLine[i + 1]);
                        step.add(new SignalValue(
                                (IntegerSignal) inputObject,
                                recentValue));
                        stepNumber = nextStepNumber + 1;
                    }
                }
            //} else if (inputObject.getClass().equals(
            //        Signal.class)) {
            } else if (inputObject instanceof Variable) {
                String recentValue =
                    ((Variable) inputObject).toExpString();
                trace.seek(-1);
                for (int i = 1; i < tokenizedLine.length; i += 2) {
                    if (stepNumber == Integer.parseInt(
                            tokenizedLine[i])) {
                        if (trace.hasNextStep()) {
                            step = trace.getNextStep();
                        } else {
                            step = new TraceStep();
                            trace.addStep(step);
                        }
                        recentValue = tokenizedLine[i + 1];
                        step.add(new VariableValue(
                                (Variable) inputObject,
                                recentValue));
                        stepNumber++;
                    } else {
                        int nextStepNumber = Integer.parseInt(
                                tokenizedLine[i]);
                        for (int j = stepNumber;
                            j < nextStepNumber; j++) {
                            if (trace.hasNextStep()) {
                                step = trace.getNextStep();
                            } else {
                                step = new TraceStep();
                            }
                            step.add(new VariableValue(
                                    (Variable) inputObject,
                                    recentValue));
                        }
                        if (trace.hasNextStep()) {
                            step = trace.getNextStep();
                        } else {
                            step = new TraceStep();
                            trace.addStep(step);
                        }
                        recentValue = tokenizedLine[i + 1];
                        step.add(new VariableValue(
                                (Variable) inputObject,
                                recentValue));
                        stepNumber = nextStepNumber + 1;
                    }
                }
            }
        }
        currentLine = datainput.readLine();
        if (currentLine == null) {
            break;
                }
            }
            currentLine = datainput.readLine();
        }
        datainput.close();
    } catch (IOException io) {
        throw new FileInterfaceException(io.getMessage());
    }
    return trace;
}

/**
 * Writes the trace specified by the parameter <code>td</code> into the
 * file specified by <code>f</code> in the M file format.
 * @param td - the trace to be saved.
 * @param f - the file (name) to save the trace into.
 * @return the file name the trace has been written into.
 * @throws FileInterfaceException if an error occurs while opening the
 * file.
 */
public File writeTraceFile(final TraceData td, final File f)
    throws FileInterfaceException {
    TraceStep step;
    ArrayList namesList;
    try {
        FileWriter dataoutput = new FileWriter(f);
        dataoutput.write("%␣This␣file␣was␣automatically␣generated␣by"
                + "␣MFileFormat.writeTraceFile\n\n");
        namesList = generateNamesList(td);
        int stepNumber;
        String recentValue;
        boolean written;

        for (int i = 0; i < namesList.size(); i++) {
            stepNumber = 0;
            recentValue = getInitialValue((String) namesList.get(i));
            td.play();
            td.startTrace();
            dataoutput.write(namesList.get(i) + "␣=␣[");
            while (td.hasNextStep()) {
                step = td.getNextStep();
                VariableValue[] varval = step.getVariableAssignments();
                SignalValue[] sigval = step.getSignalAssignments();
                written = false;

                if (stepNumber != 0) {
                    dataoutput.write(";␣");
                }

                for (int j = 0; j < sigval.length; j++) {
                    String name = ((Signal) sigval[j].getObject())
                        .getName();
                    if (namesList.get(i).equals(name)) {
                        Integer val = sigval[j].getValue();
                        recentValue = val.toString();
                        dataoutput.write(stepNumber + "␣"
                                + recentValue);
                        written = true;
                        break;
                    }
```

```java
                }

                for (int j = 0; j < varval.length; j++) {
                    String name = ((Variable) varval[j].getObject())
                        .getName();
                    if (namesList.get(i).equals(name)) {
                        String val = varval[j].getStringValue();
                        recentValue = val;
                        dataoutput.write(stepNumber + "␣" + val);
                        written = true;
                        break;
                    }
                }
                if (!written) {
                    dataoutput.write(stepNumber + "␣" + recentValue);
                }
                stepNumber++;
            }
            dataoutput.write("];\n");
            dataoutput.flush();
        }
        dataoutput.close();
        return f;
    } catch (IOException io) {
        throw new FileInterfaceException(io.getMessage());
    }
}

/**
 * Returns an instance of the M file filter.
 * @return an instance of the M file filter.
 */
public FileFilter getTraceFileFileFilter() {
    return MFileFilter.getInstance();
}

/**
 * Sets the input signals and input variables that are used to construct
 * the <code>TraceData</code> object.
 * @param in - the input variables and input signals that appear in the
 * trace file. Inputs that are not specified here will not be build into
 * the trace!!!
 */
public void setInputs(final ArrayList in) {
    inputs = in;
}

/**
 * Searches for the object that is referenced by the name in the trace
 * file and writes this into the global variable <code>inputObject</code>.
 */
private void searchForInputObject() {
    for (int i = 0; i < inputs.size(); i++) {
        if (inputs.get(i) instanceof Event) {
            if (currentLine.startsWith(((Event) inputs.get(i))
                    .getName())) {
                inputObject = inputs.get(i);
                return;
            } else {
                inputObject = null;
            }
        } else if (inputs.get(i) instanceof Variable) {
            if (currentLine.startsWith(((Variable) inputs.get(i))
                    .getName())) {
                inputObject = inputs.get(i);
                return;
            } else {
                inputObject = null;
            }
        }
    }
}

/**
 * Generates a list of names of objects referenced in the trace.
 * @param td - the trace for which the names list has to be created.
 * @return a list of all names that appear in the trace.
 */
private ArrayList generateNamesList(final TraceData td) {
    TraceStep step;
    ArrayList namesList = new ArrayList();
    td.play();
    td.startTrace();

    while (td.hasNextStep()) {
        step = td.getNextStep();
        VariableValue[] varval = step.getVariableAssignments();
        SignalValue[] sigval = step.getSignalAssignments();

        for (int i = 0; i < sigval.length; i++) {
            if (!namesList.contains(((Signal) sigval[i].getObject())
                    .getName())) {
                namesList.add(((Signal) sigval[i].getObject())
                        .getName());
            }
        }

        for (int i = 0; i < varval.length; i++) {
            if (!namesList.contains(((Variable) varval[i].getObject())
                    .getName())) {
                namesList.add(((Variable) varval[i].getObject())
                    .getName());
            }
        }
    }
    return namesList;
}

/**
 * Gets the initial value of the variable specified by <code>obj</code>.
 * @param obj - the variable whose initial value is of interest.
 * @return the initial value of the variable specified by the parameter.
 */
private String getInitialValue(final String obj) {
    Iterator it = inputs.iterator();
    Object next;
    while (it.hasNext()) {
        next = it.next();
        if (next instanceof Variable) {
            if (((Variable) next).getName().equals(obj)) {
```

```
            return ((Variable) next).toExpString();
        }
    }
}
```

```
            return new String("0");
        }
    }
```

B.4. kiel.fileInterface.ESI

B.4.1. ESIFileFilter

```
//$Id:
package kiel.fileInterface.ESI;

import java.io.File;

import javax.swing.filechooser.FileFilter;


/**
 * @author apo
 * @version $Revision: 1.1 $ last modified $Date: 2005/08/11 12:41:40 $
 */
public class ESIFileFilter extends FileFilter {

    /**An instance counter.*/
    private static ESIFileFilter instance = null;

    /**Ensures that only one instance exists.*/
    public static ESIFileFilter getInstance(){
        if(instance == null){
            instance = new ESIFileFilter();
        }
        return instance;
    }


        public boolean accept(File f) {
                if(f.isDirectory()){
            return true;
        }
        return (
                f.canRead() && f.isFile()
                && f.getPath().toLowerCase().endsWith(".esi"));
    }


        public String getDescription() {
                return new String("ESI␣Trace␣File");
        }
}
```

B.4.2. ESIFileFormat

```java
//$Id:
package kiel.fileInterface.ESI;

import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.LineNumberReader;
import java.util.ArrayList;

import javax.swing.filechooser.FileFilter;

import kiel.configMngr.SignalValue;
import kiel.configMngr.VariableValue;
import kiel.dataStructure.Variable;
import kiel.dataStructure.eventexp.Event;
import kiel.dataStructure.eventexp.IntegerSignal;
import kiel.dataStructure.eventexp.Signal;
import kiel.fileInterface.FileInterfaceException;
import kiel.fileInterface.TraceFileInterface;
import kiel.simulationTrace.TraceData;
import kiel.simulationTrace.TraceStep;

/**
 * @author apo
 * @version $Revision: 1.4 $ last modified $Date: 2005/12/15 13:31:12 $
 */
public final class ESIFileFormat extends TraceFileInterface {

    /**
     * Input variables and input signals.
     */
    private ArrayList inputs;

    /**
     * The string that is currently processed.
     */
    private String currentLine;

    /**
     * The object that is currently added to the trace.
     */
    private Object inputObject;

    /**
     * Reads the file specified by the parameter <code>f</code> and creates
     * a <code>TraceData</code> object according to the contents of the file.
     * @param f - the file to be read.
     * @return the trace that is stored in the file.
     * @throws FileInterfaceException if an error occurs while opening the
     * file.
     */
    public TraceData readTraceFile(final File f)
        throws FileInterfaceException {
        TraceData trace = new TraceData();
        TraceStep step;
        try {
            FileReader datareader = new FileReader(f);
            LineNumberReader datainput = new LineNumberReader(datareader);
            currentLine = datainput.readLine();
            step = new TraceStep();

            while (currentLine != null) {
                while ((!currentLine.trim().endsWith(";"))
                        && (!currentLine.trim().equals(""))
                        && (!currentLine.startsWith("%"))) {
                    searchForInputObject();
                    if (inputObject != null) {
                        if (inputObject.getClass().equals(
                                IntegerSignal.class)) {
                            if (currentLine.indexOf('=') != -1) {
                                step.add(new SignalValue(
                                        (IntegerSignal) inputObject,
                                        new Integer((currentLine.substring(
                                                currentLine.indexOf('=')
                                                + 1).trim()))));
                            } else {
                                step.add(new SignalValue(
                                        (IntegerSignal) inputObject,
                                        null));
                            }
                        } else if (inputObject.getClass().equals(
                                Signal.class)) {
                            step.add((Signal) inputObject);
                        } else if (inputObject instanceof Variable) {
                            if (currentLine.indexOf('=') != -1) {
                                step.add(new VariableValue(
                                        (Variable) inputObject,
                                        (currentLine.substring(
                                                currentLine.indexOf('=')
                                                + 1).trim())));
                            } else {
                                step.add((Variable) inputObject);
                            }
                        } else {
                            step.add((Event) inputObject);
                        }
                    }
                    currentLine = datainput.readLine();
                }
                if (currentLine.trim().endsWith(";")) {
                     trace.addStep(step);
                     step = new TraceStep();
                }
                currentLine = datainput.readLine();
            }
            datainput.close();
        } catch (IOException io) {
            throw new FileInterfaceException(io.getMessage());
        }
        return trace;
    }
```

```
/**
 * Writes the trace specified by the parameter <code>td</code> into the
 * file specified by <code>f</code> in the ESI file format.
 * @param td - the trace to be saved.
 * @param f - the file (name) to save the trace into.
 * @return the file name the trace has been written into.
 * @throws FileInterfaceException if an error occurs while opening the
 * file.
 */
public File writeTraceFile(final TraceData td, final File f)
        throws FileInterfaceException {
    TraceStep step;
    try {
        FileWriter dataoutput = new FileWriter(f);
        dataoutput.write("%␣This␣file␣was␣automatically␣generated␣by"
                + "␣ESIFileFormat.writeTraceFile\n\n");
        td.play();
        td.startTrace();
        int stepNumber = 1;

        while (td.hasNextStep()) {
            dataoutput.write("%␣STEP␣" + stepNumber + "\n");
            step = td.getNextStep();
            Event[] ev = step.getEvents();
            Signal[] sg = step.getSignals();
            IntegerSignal[] intsig = step.getValuedSignals();
            VariableValue[] varval = step.getVariableAssignments();
            SignalValue[] sigval = step.getSignalAssignments();

            for (int i = 0; i < ev.length; i++) {
                dataoutput.write(ev[i].getName());
                dataoutput.write("\n");
                dataoutput.flush();
            }

            for (int i = 0; i < sg.length; i++) {
                dataoutput.write(sg[i].getName());
                dataoutput.write("\n");
                dataoutput.flush();
            }

            for (int i = 0; i < intsig.length; i++) {
                dataoutput.write(intsig[i].getName());
                dataoutput.write("\n");
                dataoutput.flush();
            }

            for (int i = 0; i < sigval.length; i++) {
                Integer val = sigval[i].getValue();
                dataoutput.write(((Signal) sigval[i].getObject())
                        .getName());
                if (val != null) {
                    dataoutput.write("␣=␣");
                    dataoutput.write(val.toString());
                }
                dataoutput.write("\n");
                dataoutput.flush();
            }
            for (int i = 0; i < varval.length; i++) {
                String val = varval[i].getStringValue();
                dataoutput.write(((Variable) varval[i].getObject())
                        .getName());
                if (val != null) {
                    dataoutput.write("␣=␣");
                    dataoutput.write(val);
                }
                dataoutput.write("\n");
                dataoutput.flush();
            }

            dataoutput.write(";\n");
            dataoutput.flush();
            stepNumber++;
        }
        dataoutput.close();
        return f;
    } catch (IOException io) {
        throw new FileInterfaceException(io.getMessage());
    }
}

/**
 * Returns an instance of the ESI file filter.
 * @return an instance of the ESI file filter.
 */
public FileFilter getTraceFileFileFilter() {
    return ESIFileFilter.getInstance();
}

/**
 * Sets the input signals and input variables that are used to construct
 * the <code>TraceData</code> object.
 * @param in - the input variables and input signals that appear in the
 * trace file. Inputs that are not specified here will not be build into
 * the trace!!!
 */
public void setInputs(final ArrayList in) {
    inputs = in;
}

/**
 * Searches for the object that is referenced by the name in the trace
 * file and writes this into the global variable <code>inputObject</code>.
 */
private void searchForInputObject() {
    for (int i = 0; i < inputs.size(); i++) {
        if (inputs.get(i) instanceof Event) {
            if (currentLine.startsWith(((Event) inputs.get(i)).getName())) {
                inputObject = inputs.get(i);
                return;
            } else {
                inputObject = null;
            }
        } else if (inputs.get(i) instanceof Variable) {
            if (currentLine.startsWith(((Variable) inputs.get(i))
                    .getName())) {
                inputObject = inputs.get(i);
                return;
        } else {
            inputObject = null;
        }
    }
```

```
            }
        }
    }
```

B.5. kiel.util

B.5.1. MatlabStateflowGrabber

```java
//$Id: MatlabStateflowGrabber.java,v 1.24 2005/12/12 16:58:56 spr Exp $
package kiel.util;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Arrays;
import java.util.HashMap;
import java.util.LinkedList;
import java.util.ListIterator;
import java.util.ArrayList;

import kiel.dataStructure.ANDState;
import kiel.dataStructure.CompositeState;
import kiel.dataStructure.Constant;
import kiel.dataStructure.DelimiterLine;
import kiel.dataStructure.History;
import kiel.dataStructure.InitialArc;
import kiel.dataStructure.InitialState;
import kiel.dataStructure.Junction;
import kiel.dataStructure.Node;
import kiel.dataStructure.ORState;
import kiel.dataStructure.Priority;
import kiel.dataStructure.Region;
import kiel.dataStructure.SimpleState;
import kiel.dataStructure.State;
import kiel.dataStructure.StateChart;
import kiel.dataStructure.StringLabel;
import kiel.dataStructure.StringVariable;
import kiel.dataStructure.Transition;
import kiel.dataStructure.TransitionLabel;
import kiel.dataStructure.Variable;
import kiel.dataStructure.action.ActionsStringLabel;
import kiel.dataStructure.boolexp.BooleanExpression;
import kiel.dataStructure.boolexp.BooleanFalse;
import kiel.dataStructure.boolexp.BooleanTrue;
import kiel.dataStructure.boolexp.BooleanVariable;
import kiel.dataStructure.doubleexp.DoubleConstant;
import kiel.dataStructure.doubleexp.DoubleVariable;
import kiel.dataStructure.eventexp.Event;
import kiel.dataStructure.eventexp.IntegerSignal;
import kiel.dataStructure.floatexp.FloatConstant;
import kiel.dataStructure.floatexp.FloatVariable;
import kiel.dataStructure.int16exp.Integer16Constant;
import kiel.dataStructure.int16exp.Integer16Variable;
import kiel.dataStructure.int8exp.Integer8Constant;
import kiel.dataStructure.int8exp.Integer8Variable;
import kiel.dataStructure.intexp.IntegerConstant;
import kiel.dataStructure.intexp.IntegerVariable;
import kiel.dataStructure.uint16exp.UInt16Constant;
import kiel.dataStructure.uint16exp.UInt16Variable;
import kiel.dataStructure.uint32exp.UInt32Constant;
import kiel.dataStructure.uint32exp.UInt32Variable;
import kiel.dataStructure.uint8exp.UInt8Constant;
import kiel.dataStructure.uint8exp.UInt8Variable;
import kiel.graphicalInformations.CompositeStateLayoutInformation;
import kiel.graphicalInformations.DelimiterLineLayoutInformation;
import kiel.graphicalInformations.EdgeLayoutInformation;
import kiel.graphicalInformations.LabelLayoutInformation;
import kiel.graphicalInformations.MovetoPath;
import kiel.graphicalInformations.NodeLayoutInformation;
import kiel.graphicalInformations.Point;
import kiel.graphicalInformations.Properties;
import kiel.graphicalInformations.QuadraticCurvetoPath;
import kiel.graphicalInformations.View;
/**
 * @author Adrian Posor (apo)
 * @version $Revision: 1.24 $ last modified $Date: 2005/12/12 16:58:56 $
 */
public final class MatlabStateflowGrabber {
    /**
     * The file name of the currently processed chart.
     */
    public static String currentFileName = "";

    /**
     * The name of the currently processed chart.
     */
    public static String currentChartName = "";

    /**
     * The hash map used to map between created objects of the data structure
     * and objects in the Stateflow chart.
     */
    public static HashMap map = new HashMap();

    /**
     * The Matlab prompt, usually '>> '.
     */
    private static String matlabPrompt;

    /**
     * The string that indicates an error in Matlab, usually '???'.
     */
    private static String matlabErrorIndicator;

    /**
     * The buffered reader used to read Matlab's output.
     */
    private static BufferedReader fromMatlab;
```

```
/**
 * The buffered writer used to send commands to Matlab.
 */
private static BufferedWriter toMatlab;

/**
 * The view which includes the graphical information (layout) of the
 * currently processed chart.
 */
private static View actualView;

/**
 * The cooridnates of the chart's boundary.
 */
private static int[] chartBoundary = new int[2];

/**
 * The factor used to scale the chart during import.
 */
private static float chartScale =
    MatlabStateflowProperties.getChartScale();

/**
 * The last answer of Matlab.
 */
private static String input = "";

/**
 * The file the log is written into.
 */
private static LogFile log = null;

/**
 * A flag that indicates whether the currently processed chart contains
 * any subcharts.
 */
private static boolean containsSubcharts = false;

/**
 * The command to obtain the Stateflow root object.
 */
private static final String GET_ROOT_COMMAND = "rt=sfroot";

/**
 * The command to obtain the Simulink Model.
 */
private static final String GET_MODELS_COMMAND =
    "model=rt.find('-isa','Simulink.BlockDiagram')";

/**
 * The command to obtain the Stateflow chart.
 */
private static final String GET_CHART_COMMAND =
    "chart=model.find('-isa','Stateflow.Chart')";

/**
 * The command to obtain states which are direct children of their parent.
 */
private static final String GET_STATES_COMMAND =
    ".find('-isa','Stateflow.State','-depth',1)";

/**
 * The command to obtain transitions which are direct children of their
 * parent state.
 */
private static final String GET_TRANSITIONS_COMMAND =
    ".find('-isa','Stateflow.Transition','-depth',1)";

/**
 * The command to obtain junctions which are direct children of their
 * parent state.
 */
private static final String GET_JUNCTIONS_COMMAND =
    ".find('-isa','Stateflow.Junction','-depth',1)";

/**
 * The command to obtain local events.
 */
private static final String GET_EVENTS_COMMAND =
    ".find('-isa','Stateflow.Event','-depth',1)";

/**
 * The command to obtain input events.
 */
private static final String GET_INPUTEVENTS_COMMAND =
    ".find('-isa','Stateflow.Event','-depth',1,'Scope','Input')";

/**
 * The command to obtain output events.
 */
private static final String GET_OUTPUTEVENTS_COMMAND =
    ".find('-isa','Stateflow.Event','-depth',1,'Scope','Output')";

/**
 * The command to obtain local events.
 */
private static final String GET_LOCALEVENTS_COMMAND =
    ".find('-isa','Stateflow.Event','-depth',1,'Scope','Local')";

/**
 * The command to obtain variables.
 */
private static final String GET_VARIABLES_COMMAND =
    ".find('-isa','Stateflow.Data','-depth',1)";

/**
 * The command to obtain the editor object of the currently
 * processed chart.
 */
private static final String GET_EDITOR_COMMAND = "editor=chart.editor";

/**
 * The suffix to obtain the initial value of a variable.
 */
private static final String INITIAL_VALUE = ".props.InitialValue";

/**
 * The suffix to obtain the type of a variable.
 */
private static final String DTYPE = ".DataType";
```

```
/**
 * The suffix to obtain the identifier of a Stateflow object.
 */
private static final String ID = ".Id";

/**
 * The suffix to obtain the name of a Stateflow object.
 */
private static final String NAME = ".Name";

/**
 * The suffix to obtain the label of a state of transition.
 */
private static final String LABEL = ".LabelString";

/**
 * The suffix to obtain the position of a transition label.
 */
private static final String LABEL_POS = ".LabelPosition";

/**
 * The suffix to obtain the position of a graphical Stateflow object.
 */
private static final String POSITION = ".Position";

/**
 * The suffix to obtain the center cooridinates of a junction.
 */
private static final String JUNC_POS = ".Position.Center";

/**
 * The suffix to obtain the radius of a junction.
 */
private static final String JUNC_RADIUS = ".Position.Radius";

/**
 * The suffix to obtain the midpoint of a transition.
 */
private static final String TR_MIDPOINT = ".MidPoint";

/**
 * The suffix to obtain the position and size of
 * the window that contains the chart.
 */
private static final String CHART_SIZE = ".WindowPosition";

/**
 * The suffix to obtain the source of a transition.
 */
private static final String SRC = ".Source";

/**
 * The suffix to obtain the source end point of a transition.
 */
private static final String SRC_EP = ".SourceEndPoint";

/**
 * The suffix to obtain the destination of a transition.
 */
private static final String DEST = ".Destination";

/**
 * The suffix to obtain the o'clock position of a transition source.
 */
private static final String SRC_OCLOCK = ".SourceOClock";

/**
 * The suffix to obtain the o'clock position of a transition destination.
 */
private static final String DEST_OCLOCK = ".DestinationOClock";

/**
 * The suffix to obtain the decomposition type of a state.
 */
private static final String DECOMPO = ".Decomposition";

/**
 * The suffix to obtain the type of a transition (connective, history).
 */
private static final String TYPE = ".Type";

/**
 * The suffix to optain the port of an input or output event.
 */
private static final String PORT = ".Port";

/**
 * The suffix to obtain the trigger type of an input event.
 */
private static final String TRIGGER = ".Trigger";

/**
 * The suffix to check whether a state is a subchart.
 */
private static final String SUBCHART = ".IsSubchart";

/**
 * The suffix to obtain the scope of a variable (local, input, or output).
 */
private static final String SCOPE = ".Scope";

/**
 * The suffix to set the 'HasOutputData' option for states to true.
 */
private static final String ENABLE_OUTPUTDATA = ".HasOutputData=true";

/**
 * The suffix to obtain the execution order number of a transition.
 */
private static final String EXEC_ORDER = ".ExecutionOrder";

/**
 * Size of the input buffer (for receiving Matlab's answer).
 */
private static final int BUFF_SIZE = 20;

/**
 * Utility classes should not have a public or default constructor.
 */
```

```
    private MatlabStateflowGrabber() { }

    /**
     * Receives output from Matlab. This method uses a buffered reader
     * that is connected to Matlab's standard output. The stream is read
     * until a Matlab prompt is encountered. Matlab's answer is stored
     * in the global variable <code>input</code>. If an error is indicated
     * by Matlab, then an exception is thrown.
     * @throws IOException if an error occurs in the input or output stream.
     * @throws MatlabStateflowGrabberException if an error is encountered
     * by Matlab.
     */
    private static void checkMatlabResponse() throws IOException,
        MatlabStateflowGrabberException {
        input = "";
        char[] inputBuf = new char[BUFF_SIZE];
        int numRead = 0;

        while (!input.endsWith(matlabPrompt)) {
            numRead = fromMatlab.read(inputBuf, 0, inputBuf.length);
            if (numRead > 0) {
                input += new String(inputBuf, 0, numRead);
            }
        }
        log.log(0, input);
        if (input.indexOf(matlabErrorIndicator) != -1) {
            closeLogFile();
            throw new MatlabStateflowGrabberException(input);
        }
    }

    /**
     * Sends a command to Matlab and receives Matlab's response. This method
     * calls <code>checkMatlabResponse</code> to receive Matlab's answer to
     * the command.
     * @param command - the command that has to be sent to Matlab.
     * @return Matlab's answer to the command.
     * @throws IOException if an error occurs in the input or output stream.
     * @throws MatlabStateflowGrabberException if an error is encountered
     * by Matlab.
     */
    private static String[] handleMatlabCommand(final String command)
        throws IOException, MatlabStateflowGrabberException {
        log.log(0, "Handle␣Matlab␣command:␣" + command);
        toMatlab.write(command);
        toMatlab.newLine();
        toMatlab.flush();
        checkMatlabResponse();
        return input.trim().split("\\s+");
    }

    /**
     * Obtains the size of variable (array) <code>s</code> in the
     * Matlab workspace.
     * @param s - the variable (array) in the Matlab workspace one wants
     *  to know the size of.
     *  @return the size of the variable, that is the number of elements
     *  in the array.
     * @throws IOException if an error occurs in the input or output stream.
     * @throws MatlabStateflowGrabberException if an error is encountered
     * by Matlab.
     */
    private static int getArraySizeOf(final String s) throws IOException,
        MatlabStateflowGrabberException {
        String[] answer = handleMatlabCommand("size(" + s + ")");
        return Integer.parseInt(answer[2]);
    }

    /**
     * Obtains the number of transitions that have been previously
     * obtained.
     * @return the number of previously obtained transitions.
     * @throws IOException if an error occurs in the input or output stream.
     * @throws MatlabStateflowGrabberException if an error is encountered
     * by Matlab.
     */
    private static int getNumberOfTransitions() throws IOException,
        MatlabStateflowGrabberException {
        String[] answer = handleMatlabCommand("transitions.size");
        return Integer.parseInt(answer[2]);
    }

    /**
     * Obtains the number of junctions that have been previoulsy
     * obtained.
     * @return the number of junctions previously obtained.
     * @throws IOException if an error occurs in the input or output stream.
     * @throws MatlabStateflowGrabberException if an error is encountered
     * by Matlab.
     */
    private static int getNumberOfJunctions() throws IOException,
        MatlabStateflowGrabberException {
        String[] answer = handleMatlabCommand("junctions.size");
        return Integer.parseInt(answer[2]);
    }

    /**
     * Obtains the data type of the variable specified by
     * the <code>var</code> argument.
     * @param var - the variable one wants to know the data type of
     * @return the data type of the variable (int32, int16, int8,
     * single, double, boolean, unsigned integer (32, 16, 8 bit).
     * @throws IOException if an error occurs in the input or output stream.
     * @throws MatlabStateflowGrabberException if an error is encountered
     * by Matlab.
     */
    private static String getDataType(final String var) throws IOException,
        MatlabStateflowGrabberException {
        String[] answer = handleMatlabCommand(var + DTYPE);
        return answer[2];
    }

    /**
     * Obtains the scope of the variable specified by
     * the <code>var</code> argument.
     * @param var - the variable one wants to know the scope of.
     * @return the scope of the variable (input, output, local).
     * @throws IOException if an error occurs in the input or output stream.
     * @throws MatlabStateflowGrabberException if an error is encountered
     * by Matlab.
     */
    private static String getDataScope(final String var) throws IOException,
        MatlabStateflowGrabberException {
        String[] answer = handleMatlabCommand(var + SCOPE);
        return answer[2];
    }

    /**
     * Obtains the name of the Stateflow object specified by
     * the <code>obj</code> argument.
     * @param obj - the object one wants to know the name of
     * @return the name of the object.
     * @throws IOException if an error occurs in the input or output stream.
     * @throws MatlabStateflowGrabberException if an error is encountered
     * by Matlab.
     */
    private static String getObjectName(final String obj) throws IOException,
        MatlabStateflowGrabberException {
        log.log(0, "Get␣object␣name:␣" + obj + NAME);
        toMatlab.write(obj + NAME);
        toMatlab.newLine();
        toMatlab.flush();
        checkMatlabResponse();
        String[] answer = input.trim().split("\\n+");
        return answer[1];
    }

    /**
     * Obtains the label of a state or of a transition specified by
     * the <code>obj</code> argument.
     * @param obj - the state or transition one wants to get the label of.
     * @return the label of obj.
     * @throws IOException if an error occurs in the input or output stream.
     * @throws MatlabStateflowGrabberException if an error is encountered
     * by Matlab.
     */
    private static String getObjectLabel(final String obj) throws IOException,
        MatlabStateflowGrabberException {
        log.log(0, "Get␣object␣label:␣" + obj + LABEL);
        toMatlab.write(obj + LABEL);
        toMatlab.newLine();
        toMatlab.flush();
        checkMatlabResponse();
        /*String[] answer = input.trim().split("\\n+");
        if (answer[1].equals("?")) {
            return new String();
        }*/
        String answer = input.trim();
        answer = answer.substring(7, answer.length() - 4);
        if (answer.equals("?")) {
            return new String();
        }
        return answer;
    }

    /**
     * Obtains the position of the label of the object specified by
     * the <code>obj</code> argument.
     * @param obj - the object (state or transition) for which one
     * wants to know the label position [x, y, width, height].
     * @return the coordinates of the label of the object obj.
     * @throws IOException if an error occurs in the input or output stream.
     * @throws MatlabStateflowGrabberException if an error is encountered
     * by Matlab.
     */
    private static int[] getObjectLabelPosition(final String obj)
        throws IOException, MatlabStateflowGrabberException {
        String[] answer = handleMatlabCommand(obj + LABEL_POS);
        int[] pos = new int[4];
        if (answer.length > 7) {
            float scale = Float.parseFloat(answer[2]);

            pos[0] = (int) (Float.parseFloat(answer[4]) * scale * chartScale);
            pos[1] = (int) (Float.parseFloat(answer[5]) * scale * chartScale);
            pos[2] = (int) (Float.parseFloat(answer[6]) * scale * chartScale);
            pos[3] = (int) (Float.parseFloat(answer[7]) * scale * chartScale);
            return pos;
        }
        pos[0] = (int) (Float.parseFloat(answer[2]) * chartScale);
        pos[1] = (int) (Float.parseFloat(answer[3]) * chartScale);
        pos[2] = (int) (Float.parseFloat(answer[4]) * chartScale);
        pos[3] = (int) (Float.parseFloat(answer[5]) * chartScale);
        return pos;
    }

    /**
     * Obtains the position of the object specified by
     * the argument <code>obj</code>.
     * @param obj - the object of which one wants to know the position.
     * @return the coordinates of the object [x, y ,width, height].
     * @throws IOException if an error occurs in the input or output stream.
     * @throws MatlabStateflowGrabberException if an error is encountered
     * by Matlab.
     */
    private static int[] getObjectPosition(final String obj)throws IOException,
        MatlabStateflowGrabberException {
        String[] answer = handleMatlabCommand(obj + POSITION);
        int[] pos = new int[4];
        if (answer.length > 7) {
            float scale = Float.parseFloat(answer[2]);

            pos[0] = (int) (Float.parseFloat(answer[4]) * scale * chartScale);
            pos[1] = (int) (Float.parseFloat(answer[5]) * scale * chartScale);
            pos[2] = (int) (Float.parseFloat(answer[6]) * scale * chartScale);
            pos[3] = (int) (Float.parseFloat(answer[7]) * scale * chartScale);
            return pos;
        }
        pos[0] = (int) (Float.parseFloat(answer[2]) * chartScale);
        pos[1] = (int) (Float.parseFloat(answer[3]) * chartScale);
        pos[2] = (int) (Float.parseFloat(answer[4]) * chartScale);
        pos[3] = (int) (Float.parseFloat(answer[5]) * chartScale);
        if (chartBoundary[0] < (pos[0] + pos[2])) {
            chartBoundary[0] = (pos[0] + pos[2]);
        }
        if (chartBoundary[1] < (pos[1] + pos[3])) {
            chartBoundary[1] = (pos[1] + pos[3]);
        }
        return pos;
    }
    /**
     * Obtains the size of the currently processed chart.
     * @return the size of the currently processed chart [width, height].
     * @throws IOException if an error occurs in the input or output stream.
     * @throws MatlabStateflowGrabberException if an error is encountered
     * by Matlab.
     */
    private static int[] getChartSize() throws IOException,
        MatlabStateflowGrabberException {
        log.log(0, "Get␣chart␣size:␣");
        toMatlab.write(GET_EDITOR_COMMAND);
        toMatlab.newLine();
        toMatlab.flush();
        checkMatlabResponse();
        String[] answer = handleMatlabCommand("editor" + CHART_SIZE);
        int[] size = new int[2];
        size[0] = (int) (Float.parseFloat(answer[4]) * chartScale);
        size[1] = (int) (Float.parseFloat(answer[5]) * chartScale);
        return size;
    }

    /**
     * Obtains the unique identifier of the Stateflow object
     * specified by <code>obj</code>.
     * @param obj - the object of which one wants to know the id.
     * @return the id of obj.
     * @throws IOException if an error occurs in the input or output stream.
     * @throws MatlabStateflowGrabberException if an error is encountered
     * by Matlab.
     */
    private static Integer getObjectID(final String obj)
        throws IOException, MatlabStateflowGrabberException {
        String[] answer = handleMatlabCommand(obj + ID);
        return new Integer(answer[2]);
    }

    /**
     * Obtains the source of the transition specified by <code>trans</code>.
     * @param trans - the transition of which one wants to know the source.
     * @return the id of the source of the transition <code>trans</code>.
     * @throws IOException if an error occurs in the input or output stream.
     * @throws MatlabStateflowGrabberException if an error is encountered
     * by Matlab.
     */
    private static Integer getTransitionSource(final String trans)
        throws IOException, MatlabStateflowGrabberException {
        String[] answer = handleMatlabCommand("source=" + trans + SRC);
        if (answer[2].equals("[]")) {
//          this transition has got no source: initial
            return new Integer(-1);
        }
        return getObjectID("source");
    }

    /**
     * Obtains the destination of the transition specified
     * by <code>trans</code>.
     * @param trans - the transition of which one wants to know
     * the destination.
     * @return the id of the destination of the transition <code>trans</code>.
     * @throws IOException if an error occurs in the input or output stream.
     * @throws MatlabStateflowGrabberException if an error is encountered
     * by Matlab.
     */
    private static Integer getTransitionDestination(final String trans)
        throws IOException, MatlabStateflowGrabberException {
        log.log(0, "Get␣trans␣dest:␣" + trans + DEST);
        toMatlab.write("dest=" + trans + DEST);
        toMatlab.newLine();
        toMatlab.flush();
        checkMatlabResponse();
        return getObjectID("dest");
    }

    /**
     * Obtains the source end point of the transition specified
     * by <code>trans</code>.
     * @param trans - the transition of which one wants to know
     * the source end point.
     * @return the source end point [x, y].
     * @throws IOException if an error occurs in the input or output stream.
     * @throws MatlabStateflowGrabberException if an error is encountered
     * by Matlab.
     */
    private static int[] getTransitionSourceEndPoint(final String trans)
        throws IOException, MatlabStateflowGrabberException {
        String[] answer = handleMatlabCommand(trans + SRC_EP);
        int[] sep = new int[2];
        sep[0] = (int) (Float.parseFloat(answer[2]) * chartScale);
        sep[1] = (int) (Float.parseFloat(answer[3]) * chartScale);
        return sep;
    }

    /**
     * Obtains the midpoint of the transition specified by <code>trans</code>.
     * @param trans - the transition of which one wants to know the midpoint.
     * @return the midpoint [x, y].
     * @throws IOException if an error occurs in the input or output stream.
     * @throws MatlabStateflowGrabberException if an error is encountered
     * by Matlab.
     */
    private static int[] getTransitionMidPoint(final String trans)
        throws IOException, MatlabStateflowGrabberException {
        String[] answer = handleMatlabCommand(trans + TR_MIDPOINT);
        int[] pos = new int[2];
        pos[0] = (int) (Float.parseFloat(answer[2]) * chartScale);
        pos[1] = (int) (Float.parseFloat(answer[3]) * chartScale);
        return pos;
    }

    /**
     * Obtains the destination o'clock value of the transition specified
     * by <code>trans</code> and calculates the coordinates of the
     * destination end point.
     * @param trans - the transition for which one wants to know the
     * destination end point.
     * @param destState - the state that is the destination of the transition
     * specified by <code>trans</code>.
     * @return the destination end point [x, y].
     * @throws IOException if an error occurs in the input or output stream.
```

```java
     * @throws MatlabStateflowGrabberException if an error is encountered
     * by Matlab.
     */
    private static int[] getTransitionDestinationEndPoint(final String trans,
                                        final Node destState)
                                        throws IOException,
                                        MatlabStateflowGrabberException {
        int[] intersection = new int[2];
        Point pIntersect = null;
        NodeLayoutInformation nli = actualView.getLayoutInformation(destState);
        int[] pos = {nli.getXPos(), nli.getYPos(), nli.getWidth(),
                    nli.getHeight()};
        Point corner1 = new Point(pos[0], pos[1]);
        Point corner2 = new Point(pos[0] + pos[2], pos[1]);
        Point corner3 = new Point(pos[0] + pos[2], pos[1] + pos[3]);
        Point corner4 = new Point(pos[0], pos[1] + pos[3]);
        Point center = new Point(pos[0] + pos[2] / 2, pos[1] + pos[3] / 2);
        float oclock = getDestinationOClock(trans);
        double oclockRad = (2 * Math.PI / 12) * oclock;
        double c = Math.sqrt((double) ((pos[2] / 2) * (pos[2] / 2)
                            + (pos[3] / 2) * (pos[3] / 2)));
        double alpha1 = Math.asin((double) (pos[2] / 2) / c);
        double alpha2 = ((2 * Math.PI - 4 * alpha1) / 2) + alpha1;
        double alpha3 = alpha2 + 2 * alpha1;
        double alpha4 = 2 * Math.PI - alpha1;

        int x = (int) (Math.sin(oclockRad) * 100); x = x + center.getX();
        int y = (int) (Math.cos(oclockRad) * 100); y = -y + center.getY();
        Point p2 = new Point(x, y);
        Line direction = new Line(center, p2);
        Line line = null;

        if ((oclockRad <= alpha1) || (oclockRad > alpha4)) {
            line = new Line(corner1, corner2);
        } else if ((oclockRad <= alpha2) && (oclockRad > alpha1)) {
            line = new Line(corner2, corner3);
        } else if ((oclockRad <= alpha3) && (oclockRad > alpha2)) {
            line = new Line(corner3, corner4);
        } else {
            line = new Line(corner4, corner1);
        }
        try {
            pIntersect = direction.intersect(line);
        } catch (LinesDoNotIntersectException ldni) {
            log.log(0, ldni.toString());
        }
        intersection[0] = pIntersect.getX();
        intersection[1] = pIntersect.getY();
        return intersection;
    }

    /**
     * Obtains the destination o'clock value of the transition specified
     * by <code>trans</code>.
     * @param trans - the transition for which one wants to know the
     * destination o'clock value.
     * @return the destination o'clock value of <code>trans</code>.
     * @throws IOException if an error occurs in the input or output stream.
     * @throws MatlabStateflowGrabberException if an error is encountered
     * by Matlab.
     */
    private static float getDestinationOClock(final String trans)
        throws IOException, MatlabStateflowGrabberException {
        String[] answer = handleMatlabCommand(trans + DEST_OCLOCK);
        return Float.parseFloat(answer[2]);
    }

    /**
     * Obtains the number for transition <code>trans</code>
     * in the execution order for its source.
     * @param trans - the transition for which one wants to know
     * the number in the execution order.
     * @return the number in the execution order.
     * @throws IOException if an error occurs in the input or output stream.
     * @throws MatlabStateflowGrabberException if an error is encountered
     * by Matlab.
     */
    private static int getTransitionExecutionOrder(final String trans)
        throws IOException, MatlabStateflowGrabberException {
        String[] answer = handleMatlabCommand(trans + EXEC_ORDER);
        return Integer.parseInt(answer[2]);
    }

    /**
     * Determines whether the state specified by the argument
     * <code>state</code> is an OR state.
     * @param state - the state for which one wants to know whether it is
     * an OR state
     * @return <code>true</code> if <code>state</code> is an OR state,
     * <code>false</code> otherwise.
     * @throws IOException if an error occurs in the input or output stream.
     * @throws MatlabStateflowGrabberException if an error is encountered
     * by Matlab.
     */
    private static boolean isORState(final String state) throws IOException,
        MatlabStateflowGrabberException {
        String[] answer = handleMatlabCommand(state + DECOMPO);
        return answer[2].equals("EXCLUSIVE_OR");
    }

    /**
     * Determines whether the state specified by the argument
     * <code>state</code> is an AND state.
     * @param state - the state for which one wants to know whether it is
     * an AND state.
     * @return <code>true</code> if <code>state</code> is an AND state,
     * <code>false</code> otherwise.
     * @throws IOException if an error occurs in the input or output stream.
     * @throws MatlabStateflowGrabberException if an error is encountered
     * by Matlab.
     */
    private static boolean isANDState(final String state) throws IOException,
        MatlabStateflowGrabberException {
        String[] answer = handleMatlabCommand(state + DECOMPO);
        return answer[2].equals("PARALLEL_AND");
    }

    /**
     * Determines whether the state specified by the argument
     * <code>state</code> is a region.
     * @param state - the state for which one wants to know whether it is
     * a region.
     * @return <code>true</code> if <code>state</code> is a region,
     * <code>false</code> otherwise.
     * @throws IOException if an error occurs in the input or output stream.
     * @throws MatlabStateflowGrabberException if an error is encountered
     * by Matlab.
     */
    private static boolean isRegion(final String state) throws IOException,
        MatlabStateflowGrabberException {
        String[] answer = handleMatlabCommand(state + TYPE);
        return answer[2].equals("AND");
    }

    /**
     * Determines whether the junction specified by the argument <code>
     * junction</code> is a history junction.
     * @param junction - the junction for which one wants to know whether
     * it is a history junction.
     * @return <code>true</code> if <code>junction</code> is a history
     * junction, <code>false</code> otherwise.
     * @throws IOException if an error occurs in the input or output stream.
     * @throws MatlabStateflowGrabberException if an error is encountered
     * by Matlab.
     */
    private static boolean isHistory(final String junction) throws IOException,
        MatlabStateflowGrabberException {
        String[] answer = handleMatlabCommand(junction + TYPE);
        return answer[2].equals("HISTORY");
    }

    /**
     * Determines whether the state specified by the argument
     * <code>state</code> is a subchart.
     * @param state - the state for which one wants to know whether it
     * is a subchart.
     * @return <code>true</code> if <code>state</code> is a subchart,
     * <code>false</code> otherwise.
     * @throws IOException if an error occurs in the input or output stream.
     * @throws MatlabStateflowGrabberException if an error is encountered
     * by Matlab.
     */
    private static boolean isSubchart(final String state) throws IOException,
        MatlabStateflowGrabberException {
        String[] answer = handleMatlabCommand(state + SUBCHART);
        return answer[2].equals("1");
    }

    /**
     * Obtains the position of the junction specified by the argument
     * <code>junction</code>.
     * @param junction - the junction for which one wants to know the position.
     * @return the position of the junction specified by the argument.
     * @throws IOException if an error occurs in the input or output stream.
     * @throws MatlabStateflowGrabberException if an error is encountered
     * by Matlab.
     */
    private static int[] getJunctionPosition(final String junction)
        throws IOException, MatlabStateflowGrabberException {
        String[] answer = handleMatlabCommand(junction + JUNC_POS);
        int[] pos = new int[2];
        pos[0] = (int) (Float.parseFloat(answer[2]) * chartScale);
        pos[1] = (int) (Float.parseFloat(answer[3]) * chartScale);
        return pos;
    }

    /**
     * Obtains the radius of the junction specified by the argument
     * <code>junction</code>.
     * @param junction - the junction for which one wants to know the radius.
     * @return the radius of the specified junction.
     * @throws IOException if an error occurs in the input or output stream.
     * @throws MatlabStateflowGrabberException if an error is encountered
     * by Matlab.
     */
    private static int getJunctionRadius(final String junction)
        throws IOException, MatlabStateflowGrabberException {
        String[] answer = handleMatlabCommand(junction + JUNC_RADIUS);
        return (int) (Float.parseFloat(answer[2]) * chartScale);
    }

    /**
     * Determines whether the state specified by the argument
     * <code>state</code> is a simple state.
     * @param state - the state for which one wants to know whether it
     * is a simple state.
     * @return <code>true</code> if <code>state</code> is a simple state,
     * <code>false</code> otherwise.
     * @throws IOException if an error occurs in the input or output stream.
     * @throws MatlabStateflowGrabberException if an error is encountered
     * by Matlab.
     */
    private static boolean isSimpleState(final String state)
        throws IOException, MatlabStateflowGrabberException {
        toMatlab.write("children=" + state + GET_STATES_COMMAND);
        toMatlab.newLine();
        toMatlab.flush();
        checkMatlabResponse();
        String[] answer = handleMatlabCommand("children.size");
        return answer[2].equals("1");
    }

    /**
     * Obtains the initial value of the variable specified by the argument
     * <code>var</code> as a double.
     * @param var - the variable of which one wants to know the initial value.
     * @return the initial value of the variable <code>var</code> as a double.
     * @throws IOException if an error occurs in the input or output stream.
     * @throws MatlabStateflowGrabberException if an error is encountered
     * by Matlab.
     */
    private static double getInitialValue(final String var) throws IOException,
        MatlabStateflowGrabberException {
        String[] answer = handleMatlabCommand(var + INITIAL_VALUE);
        return Double.parseDouble(answer[2]);
    }

    /**
     * Obtains the initial value of the variable specified by the argument
     * <code>var</code> as a string.
     * @param var - the variable of which one wants to know the initial value.
     * @return the initial value of the variable <code>var</code> as a string.
     * @throws IOException if an error occurs in the input or output stream.
     * @throws MatlabStateflowGrabberException if an error is encountered
     * by Matlab.
     */
    private static String getInitialValueAsString(final String var)
        throws IOException, MatlabStateflowGrabberException {
        String[] answer = handleMatlabCommand(var + INITIAL_VALUE);
        return answer[2];
    }

    /**
     * Obtains the port number of an event or data object.
     * @param eventOrData - the event or data object of which one wants to
     * know the port number.
     * @return the port number of the specified event or data.
     * @throws IOException if an error occurs in the input or output stream.
     * @throws MatlabStateflowGrabberException if an error is encountered
     * by Matlab.
     */
    private static int getPortNumber(final String eventOrData)
        throws IOException, MatlabStateflowGrabberException {
        String[] answer = handleMatlabCommand(eventOrData + PORT);
        return Integer.parseInt(answer[2]);
    }

    /**
     * Obtains the trigger type of the event specified by the argument
     * <code>event</code>.
     * @param event - the event of which one wants to know the trigger type.
     * @return the trigger type of the specified event
     * (rising, falling, either, function).
     * @throws IOException if an error occurs in the input or output stream.
     * @throws MatlabStateflowGrabberException if an error is encountered
     * by Matlab.
     */
    private static String getTriggerType(final String event)
        throws IOException, MatlabStateflowGrabberException {
        String[] answer = handleMatlabCommand(event + TRIGGER);
        return answer[2];
    }

    /**
     * Obtains the actions of the state in Stateflow specified by the argument
     * <code>state</code> and creates equivalent actions using the KIEL
     * data structure. The actions are created in the state in KIEL specified
     * by the argument <code>stateRef</code>.
     * @param state - the Stateflow state from which one wants to
     * obtain the actions.
     * @param stateRef - the KIEL state for which the obtained actions
     * have to be created
     * @throws IOException if an error occurs in the input or output stream.
     * @throws MatlabStateflowGrabberException if an error is encountered
     * by Matlab.
     */
    private static void getAndCreateActions(final String state,
            final State stateRef)
        throws IOException, MatlabStateflowGrabberException {
        log.log(0, "Get␣and␣create␣actions:");
        toMatlab.write(state + LABEL);
        toMatlab.newLine();
        toMatlab.flush();
        checkMatlabResponse();
        String[] answer = input.trim().split("\\n+|:\\s*|;\\n*");
        for (int i = 2; i < answer.length; i += 2) {
            if (answer[i].equals("entry") || answer[i].equals("en")) {
                stateRef.setEntry(new ActionsStringLabel(answer[i + 1]));
            } else if (answer[i].equals("during") || answer[i].equals("do")) {
                stateRef.setDoActivity(new ActionsStringLabel(answer[i + 1]));
            } else if (answer[i].equals("exit") || answer[i].equals("ex")) {
                stateRef.setExit(new ActionsStringLabel(answer[i + 1]));
            } else if (answer[i].equals("bind")) {
                stateRef.setBindAction(new ActionsStringLabel(answer[i + 1]));
            } else if (answer[i].startsWith("on␣")) {
                Transition trans = new Transition();
                TransitionLabel transLabel =
                    new StringLabel(answer[i] + answer[i + 1]);
                trans.setLabel(transLabel);
                stateRef.setInternalTransition(trans);
            }
        }
    }

    /**
     * Obtains the events of the state in Stateflow specified by the argument
     * <code>state</code> and creates equivalent events using the KIEL
     * data structure. The actions are created in the state in KIEL specified
     * by the argument <code>stateRef</code>.
     * @param state - the Stateflow state from which one wants to
     * obtain the events.
     * @param stateRef - the KIEL state for which the obtained events
     * have to be created.
     * @throws IOException if an error occurs in the input or output stream.
     * @throws MatlabStateflowGrabberException if an error is encountered
     * by Matlab.
     */
    private static void getAndCreateLocalEvents(final String state,
            final State stateRef)
        throws IOException, MatlabStateflowGrabberException {
        log.log(0, "Get␣and␣create␣local␣events:");
        toMatlab.write("events=" + state + GET_EVENTS_COMMAND);
        toMatlab.newLine();
        toMatlab.flush();
        checkMatlabResponse();

        int noe = getArraySizeOf("events");
        String eventString;

        for (int eventNumber = 0; eventNumber < noe; eventNumber++) {
            eventString = "events(" + (eventNumber + 1) + ")";
            ((CompositeState) stateRef).addLocalEvent(
                    new Event(getObjectName(eventString)));
        }
    }

    /**
     * Obtains the actions of the state in Stateflow specified by the argument
     * <code>state</code> and creates equivalent actions using the KIEL
     * data structure. The actions are created in the state in KIEL specified
```

```
 * by the argument <code>stateRef</code>.
 * @param state - the Stateflow state from which one wants to
 * obtain the actions.
 * @param stateRef - the KIEL state for which the obtained actions
 * have to be created
 * @throws IOException if an error occurs in the input or output stream.
 * @throws MatlabStateflowGrabberException if an error is encountered
 * by Matlab.
 */
private static void getAndCreateLocalVariables(final String state,
        final State stateRef)
    throws IOException, MatlabStateflowGrabberException {
    log.log(0, "Get␣and␣create␣local␣variables");
    toMatlab.write("variables=" + state + GET_VARIABLES_COMMAND);
    toMatlab.newLine();
    toMatlab.flush();
    checkMatlabResponse();

    int nov = getArraySizeOf("variables");
    String variableString;
    String type;
    String scope;
    String name;
    Variable var;
    Constant con;

    for (int variableNumber = 0; variableNumber < nov; variableNumber++) {
        var = null;
        con = null;
        variableString = "variables(" + (variableNumber + 1) + ")";
        type = getDataType(variableString);
        scope = getDataScope(variableString);
        name = getObjectName(variableString);
        if (type.equals("double")) {
            con = new DoubleConstant(name,
                    getInitialValue(variableString));
            var = new DoubleVariable(name, (DoubleConstant) con);
        } else if (type.equals("single")) {
            con = new FloatConstant(name,
                    (float) getInitialValue(variableString));
            var = new FloatVariable(name, (FloatConstant) con);
        } else if (type.equals("int32")) {
            con = new IntegerConstant(name,
                    (int) getInitialValue(variableString));
            var = new IntegerVariable(name, (IntegerConstant) con);
        } else if (type.equals("int16")) {
            con = new Integer16Constant(name,
                    (short) getInitialValue(variableString));
            var = new Integer16Variable(name, (Integer16Constant) con);
        } else if (type.equals("int8")) {
            con = new Integer8Constant(name,
                    (byte) getInitialValue(variableString));
            var = new Integer8Variable(name,
                    (Integer8Constant) con);
        } else if (type.equals("uint32")) {
            con = new UInt32Constant(name,
                    (long) getInitialValue(variableString));
            var = new UInt32Variable(name, (UInt32Constant) con);
        } else if (type.equals("uint16")) {
            con = new UInt16Constant(name,
                    (int) getInitialValue(variableString));
            var = new UInt16Variable(name, (UInt16Constant) con);
        } else if (type.equals("uint8")) {
            con = new UInt8Constant(name,
                    (short) getInitialValue(variableString));
            var = new UInt8Variable(name, (UInt8Constant) con);
        } else if (type.equals("boolean")) {
            if (getInitialValueAsString(variableString).equals("true")) {
                con = new BooleanTrue(name);
                var = new BooleanVariable(name, (BooleanExpression) con);
            } else {
                con = new BooleanFalse(name);
                var = new BooleanVariable(name, (BooleanExpression) con);
            }
        } else if (type.equals("State")) {
            /* We do nothing here because we do not want to treat them as
             * output variables. They are just used to determine the
             * set of currently active states!
             */
        } else {
            /*Chart contains variable of unsupported type!*/
            var =
                new StringVariable(name
                        + getInitialValueAsString(variableString));
        }
        if (var != null) {
            if (scope.equals("Local")) {
                stateRef.addVariable(var);
            } else if (scope.equals("Constant")) {
                stateRef.addConstant(con);
            } else {
                throw new MatlabStateflowGrabberException(
                    "Chart␣contains␣variable␣of␣unsupported␣scope!");
            }
        }
    }
}

/**
 * Obtains the global variables of the chart and creates variables
 * using the KIEL data structure. The actions are created in the state
 * in KIEL specified by the argument <code>stateRef</code>.
 * @param stateRef - the KIEL root state for which the obtained variables
 * have to be created
 * @throws IOException if an error occurs in the input or output stream.
 * @throws MatlabStateflowGrabberException if an error is encountered
 * by Matlab.
 */
private static void getAndCreateVariables(final StateChart stateRef)
    throws IOException, MatlabStateflowGrabberException {
    log.log(0, "Get␣and␣create␣global␣variables");
    toMatlab.write("variables=chart" + GET_VARIABLES_COMMAND);
    toMatlab.newLine();
    toMatlab.flush();
    checkMatlabResponse();

    int nov = getArraySizeOf("variables");
    String variableString;
    String type;
    String scope;
```

```java
        String name;
        Variable var;
        Constant con;

        for (int variableNumber = 0; variableNumber < nov; variableNumber++) {
            var = null;
            con = null;
            variableString = "variables(" + (variableNumber + 1) + ")";
            type = getDataType(variableString);
            scope = getDataScope(variableString);
            name = getObjectName(variableString);
            if (type.equals("double")) {
                con = new DoubleConstant(name,
                        getInitialValue(variableString));
                var = new DoubleVariable(name, (DoubleConstant) con);
            } else if (type.equals("single")) {
                con = new FloatConstant(name,
                        (float) getInitialValue(variableString));
                var = new FloatVariable(name, (FloatConstant) con);
            } else if (type.equals("int32")) {
                con = new IntegerConstant(name,
                        (int) getInitialValue(variableString));
                var = new IntegerVariable(name, (IntegerConstant) con);
            } else if (type.equals("int16")) {
                con = new Integer16Constant(name,
                        (short) getInitialValue(variableString));
                var = new Integer16Variable(name, (Integer16Constant) con);
            } else if (type.equals("int8")) {
                con = new Integer8Constant(name,
                        (byte) getInitialValue(variableString));
                var = new Integer8Variable(name,
                        (Integer8Constant) con);
            } else if (type.equals("uint32")) {
                con = new UInt32Constant(name,
                        (long) getInitialValue(variableString));
                var = new UInt32Variable(name, (UInt32Constant) con);
            } else if (type.equals("uint16")) {
                con = new UInt16Constant(name,
                        (int) getInitialValue(variableString));
                var = new UInt16Variable(name, (UInt16Constant) con);
            } else if (type.equals("uint8")) {
                con = new UInt8Constant(name,
                        (short) getInitialValue(variableString));
                var = new UInt8Variable(name, (UInt8Constant) con);
            } else if (type.equals("boolean")) {
                if (getInitialValueAsString(variableString).equals("true")) {
                    con = new BooleanTrue(name);
                    var = new BooleanVariable(name, (BooleanExpression) con);
                } else {
                    con = new BooleanFalse(name);
                    var = new BooleanVariable(name, (BooleanExpression) con);
                }
            } else if (type.equals("State")) {
                /* We do nothing here because we do not want to treat them as
                 * output variables. They are just used to determine the
                 * set of currently active states!
                 */
            } else {
                /*Chart contains variable of unsupported type!*/
                var =
                    new StringVariable(name
                            + getInitialValueAsString(variableString));
            }
            if (var != null) {
                if (scope.equals("Local")) {
                    stateRef.addVariable(var);
                } else if (scope.equals("Input")) {
                    stateRef.addInputVariable(var);
                } else if (scope.equals("Output")) {
                    stateRef.addOutputVariable(var);
                } else if (scope.equals("Constant")) {
                    stateRef.addConstant(con);
                } else {
                    throw new MatlabStateflowGrabberException(
                        "Chart␣contains␣variable␣of␣unsupported␣scope!");
                }
            }
        }
    }

    /**
     * Tests whether the set of regions specified by the <code>regions</code>
     * argument is horizontally alignable.
     * @param regions - the set of regions which have to be tested
     * @return <code>true</code> if the given set of regions is
     * horizontally alignable, <code>false</code> otherwise.
     */
    private static boolean isHorizontallyAlignable(final State[] regions) {
        int[] prevPos = new int[4];
        int[] pos = new int[4];
        CompositeStateLayoutInformation cli;

        cli = (CompositeStateLayoutInformation)
            actualView.getLayoutInformation(regions[0]);
        prevPos[0] = cli.getXPos();
        prevPos[1] = cli.getYPos();
        prevPos[2] = cli.getWidth();
        prevPos[3] = cli.getHeight();

        for (int i = 1; i < regions.length; i++) {
            cli = (CompositeStateLayoutInformation)
                actualView.getLayoutInformation(regions[i]);
            pos[0] = cli.getXPos();
            pos[1] = cli.getYPos();
            pos[2] = cli.getWidth();
            pos[3] = cli.getHeight();
            if (((prevPos[0] + prevPos[2]) < pos[0])
                    || (pos[0] + pos[2] < prevPos[0])) {
                ;
            } else {
                return false;
            }
        }
        return true;
    }

    /**
     * Tests whether the set of regions specified by the <code>regions</code>
     * argument is vertically alignable.
     * @param regions - the set of regions which have to be tested
```

```
 * @return <code>true</code> if the given set of regions is
 * vertically alignable, <code>false</code> otherwise.
 */
private static boolean isVerticallyAlignable(final State[] regions) {
    int[] prevPos = new int[4];
    int[] pos = new int[4];
    CompositeStateLayoutInformation cli;

    cli = (CompositeStateLayoutInformation)
        actualView.getLayoutInformation(regions[0]);
    prevPos[0] = cli.getXPos();
    prevPos[1] = cli.getYPos();
    prevPos[2] = cli.getWidth();
    prevPos[3] = cli.getHeight();

    for (int i = 1; i < regions.length; i++) {
        cli = (CompositeStateLayoutInformation)
            actualView.getLayoutInformation(regions[i]);
        pos[0] = cli.getXPos();
        pos[1] = cli.getYPos();
        pos[2] = cli.getWidth();
        pos[3] = cli.getHeight();
        if (((prevPos[1] + prevPos[3]) < pos[1])
                || (pos[1] + pos[3] < prevPos[1])) {
            ;
        } else {
            return false;
        }
    }
    return true;
}

/**
 * Generates delimiter lines that seperate the specified regions.
 * @param regions - the set of regions which have to be seperated by
 * delimiter lines
 * @param parent - the state that contains the regions.
 * @param parentPos - the coordinates of the parent state.
 */
private static void generateDelimiterLines(final State[] regions,
                                           final CompositeState parent,
                                           final int[] parentPos) {
    int[] prevPos = new int[4];
    int[] pos = new int[4];
    int upperOffset;
    DelimiterLine dell;
    LinkedList verticalDels = new LinkedList();
    CompositeStateLayoutInformation cli;

    //if(isHorizontallyAlignable(regions)){
    //    Arrays.sort(regions, new StateComparatorHorizontal(actualView));
    //} else if(isVerticallyAlignable(regions)){
    //    Arrays.sort(regions, new StateComparatorVertical(actualView));
    //} else {
        Arrays.sort(regions, new StateComparator(actualView));
    //}
    upperOffset = Properties.getUpperOffset(parent);
    for (int i = 0; i < regions.length; i++) {
        cli = (CompositeStateLayoutInformation)
            actualView.getLayoutInformation(regions[i]);
```

```
        prevPos[0] = pos[0];
        prevPos[1] = pos[1];
        prevPos[2] = pos[2];
        prevPos[3] = pos[3];
        pos[0] = cli.getXPos();
        pos[1] = cli.getYPos();
        pos[2] = cli.getWidth();
        pos[3] = cli.getHeight();

        if (i != 0) {
            /*create a vertical delimiter line*/
            if ((prevPos[0] + prevPos[2]) < pos[0]) {
                dell = new DelimiterLine();
                ((ANDState) parent).addDelimiterLine(dell);
                int x1 = (pos[0] - prevPos[0]
                    - prevPos[2]) / 2 + prevPos[0] + prevPos[2];
                DelimiterLineLayoutInformation delli =
                    new DelimiterLineLayoutInformation(
                            new Point(x1, upperOffset),
                            new Point(x1, parentPos[3]));
                actualView.setLayoutInformation(dell, delli);
                verticalDels.add(dell);
            } else
                //if((pos[0] + pos[2] < prevPos[0])){
                //    dell = new DelimiterLine();
                //    ((ANDState)parent).addDelimiterLine(dell);
                //    int x1 = (prevPos[0] - pos[0] -
                //            pos[2]) / 2 + pos[0] + pos[2];
                //    DelimiterLineLayoutInformation delli =
                //        new DelimiterLineLayoutInformation(
                //                new Point(x1, UpperOffset),
                //                new Point(x1, parentPos[3]));
                //    actualView.setLayoutInformation(dell, delli);
                //    verticalDels.add(dell);
                //} else
                /*create a horizontal delimiter line*/
                if ((prevPos[1] + prevPos[3]) < pos[1]) {
                    dell = new DelimiterLine();
                    ((ANDState) parent).addDelimiterLine(dell);
                    int y1 = (pos[1] - prevPos[1]
                            - prevPos[3]) / 2 + prevPos[1] + prevPos[3];
                    DelimiterLineLayoutInformation delli =
                        new DelimiterLineLayoutInformation(
                                new Point(0, y1),
                                new Point(parentPos[2], y1));
                    actualView.setLayoutInformation(dell, delli);
                    upperOffset = y1;
                    if (verticalDels.size() > 0) {
                        calcAndSetCrossingEndPoints(verticalDels, y1);
                    }
                } //else
                //    if(pos[1] + pos[3] < prevPos[1]){
                //        dell = new DelimiterLine();
                //        ((ANDState)parent).addDelimiterLine(dell);
                //        int y1 = (prevPos[1] - pos[1] -
                //                pos[3]) / 2 + pos[1] + pos[3];
                //        DelimiterLineLayoutInformation delli =
                //            new DelimiterLineLayoutInformation(
                //                    new Point(0, y1),
                //                    new Point(parentPos[2], y1));
```

```
                    //         actualView.setLayoutInformation(dell, delli);
                    //         if(verticalDels.size() > 0){
                    //             calcAndSetCrossingEndPoints(verticalDels, y1);
                    //         }
                    //     }
            }
            cli.setUpperLeftPoint(new Point(pos[0], upperOffset));
            cli.setWidth(pos[2]);
            cli.setHeight(pos[3]);
        }
    }

    /**
     * Calculates and sets the end points of the vertical delimiter lines
     * specified by the argument <code>vdels</code> to the points of
     * intersection between the verical delimiter lines and the horizontal
     * delimiter line that goes through <code>y</code>.
     * @param vdels - the set of vertical delimiter lines for which the
     * end points have to be adapted.
     * @param y - the y coordinate of the horizontal delimiter line that
     * intersects the vertical ones.
     */
    private static void calcAndSetCrossingEndPoints(final LinkedList vdels,
            final int y) {
        ListIterator iter = vdels.listIterator(0);
        DelimiterLine del;
        DelimiterLineLayoutInformation deli;
        int x;

        while (iter.hasNext()) {
            del = (DelimiterLine) iter.next();
            deli = actualView.getLayoutInformation(del);
            x = deli.getEnd().getX();
            deli.setEnd(new Point(x, y));
        }
    }

    /**
     * Grabs all children of the currently processed state.
     * @throws IOException if an error occurs in the input or output stream.
     * @throws MatlabStateflowGrabberException if an error is encountered
     * by Matlab.
     */
    private static void grabOneLevel() throws IOException,
        MatlabStateflowGrabberException {
        log.log(0, "Grab␣one␣level:");
        /* Get all states out of Matlab level by level of containment*/
        log.log(0, "Grab␣states:");
        toMatlab.write("states=parent" + GET_STATES_COMMAND);
        toMatlab.newLine();
        toMatlab.flush();
        checkMatlabResponse();
        toMatlab.write("states=setdiff(states,␣parent)");
        toMatlab.newLine();
        toMatlab.flush();
        checkMatlabResponse();
        int nos = getArraySizeOf("states");
        int[] parentPos = new int[4];
        int[] pos = null;
        int[] prevPos = null;
        State state = null;
        State[] regions = new Region[nos];
        DelimiterLine dell = null;
        String stateString = null;
        int upperOffset = 0;

        CompositeState parent = (CompositeState) map.get(getObjectID("parent"));

        if (parent.getParent() != null) {
            parentPos = getObjectPosition("parent");
        }
        for (int stateNumber = 0; stateNumber < nos; stateNumber++) {
            stateString = "states(" + (stateNumber + 1) + ")";
            prevPos = pos;
            pos = getObjectPosition(stateString);
            if (isSubchart(stateString)) {
                containsSubcharts = true;
            }
            if ((isSimpleState(stateString) && (!isRegion(stateString)))) {
                state = new SimpleState();
                map.put(getObjectID(stateString), state);
                NodeLayoutInformation nli = new NodeLayoutInformation();
                nli.setUpperLeftPoint(new Point(pos[0]
                                    - parentPos[0],
                                    pos[1] - parentPos[1]));
                nli.setWidth(pos[2]);
                nli.setHeight(pos[3]);
                actualView.setLayoutInformation(state, nli);
            } else {
                if (isORState(stateString)) {
                    state = new ORState();
                } else {
                    state = new ANDState();
                }
                if (isRegion(stateString)) {
                    state = new Region();
                    regions[stateNumber] = state;
                }
                map.put(getObjectID(stateString), state);
                CompositeStateLayoutInformation cli =
                    new CompositeStateLayoutInformation();
                cli.setUpperLeftPoint(new Point(pos[0]
                                    - parentPos[0],
                                    pos[1] - parentPos[1]));
                cli.setWidth(pos[2]);
                cli.setHeight(pos[3]);
                actualView.setLayoutInformation(state, cli);
            }
            state.setName(getObjectName(stateString));
            parent.addSubnode(state);
            state.setParent(parent);
            getAndCreateActions(stateString, state);
            getAndCreateLocalEvents(stateString, state);
            getAndCreateLocalVariables(stateString, state);
        }
        if (regions[0] != null) {
            generateDelimiterLines(regions, parent, parentPos);
        }

        /*Get all Junctions out of Matlab level by level of containment*/
    log.log(0, "Grab␣junctions␣and␣histories:");
    toMatlab.write("junctions=parent" + GET_JUNCTIONS_COMMAND);
    toMatlab.newLine();
    toMatlab.flush();
    checkMatlabResponse();
    int nojs = getNumberOfJunctions();
    Node node = null;
    String juncString = null;

    for (int junctionNumber = 0; junctionNumber < nojs; junctionNumber++) {
        juncString = "junctions(" + (junctionNumber + 1) + ")";

        if (isHistory(juncString)) {
            node = new History();
        } else {
            node = new Junction();
        }
        parent.addSubnode(node);
        node.setParent(parent);
        map.put(getObjectID(juncString), node);
        pos = getJunctionPosition(juncString);
        int rad = getJunctionRadius(juncString);
        NodeLayoutInformation nli = new NodeLayoutInformation();
        nli.setUpperLeftPoint(new Point(pos[0] - rad - parentPos[0],
                          pos[1] - rad - parentPos[1]));
        nli.setWidth(rad * 2);
        nli.setHeight(rad * 2);
        actualView.setLayoutInformation(node, nli);
    }

    /*Get all Transitions out of Matlab level by level of containment*/
    log.log(0, "Grab␣transitions:");
    toMatlab.write("transitions=parent" + GET_TRANSITIONS_COMMAND);
    toMatlab.newLine();
    toMatlab.flush();
    checkMatlabResponse();
    int nots = getNumberOfTransitions();
    Transition trans = null;
    String transString = null;
    int[] sep;
    int[] mip;
    int[] dep;

    for (int transitionNumber = 0; transitionNumber < nots;
        transitionNumber++) {
        transString = "transitions(" + (transitionNumber + 1) + ")";
        node = (Node) map.get(getTransitionSource(transString));
        if (node == null) {
            trans = new InitialArc();
            InitialState initial = new InitialState();
            parent.addSubnode(initial);
            initial.setParent(parent);
            trans.setSource(initial);
            EdgeLayoutInformation eli = new EdgeLayoutInformation();
            actualView.setLayoutInformation(trans, eli);
            sep = getTransitionSourceEndPoint(transString);
            NodeLayoutInformation nli = new NodeLayoutInformation();
            nli.setUpperLeftPoint(new Point(sep[0] - 3 - parentPos[0],
                             sep[1] - 3 - parentPos[1]));
            nli.setWidth(6);
            nli.setHeight(6);
            actualView.setLayoutInformation(initial, nli);
        } else {
            trans = new Transition();
            trans.setSource(node);
        }
        node = (Node) map.get(getTransitionDestination(transString));
        trans.setTarget(node);
        String label = getObjectLabel(transString);
        StringLabel sl = new StringLabel(label);
        trans.setLabel(sl);
        LabelLayoutInformation lli = new LabelLayoutInformation();
        pos = getObjectLabelPosition(transString);
        lli.setUpperLeftPoint(new Point(pos[0] - parentPos[0],
                          pos[1] - parentPos[1]));
        lli.setDimension(label);
        actualView.setLayoutInformation(sl, lli);
        sep = getTransitionSourceEndPoint(transString);
        mip = getTransitionMidPoint(transString);
        dep = getTransitionDestinationEndPoint(transString, node);
        Priority prio = new Priority(
                getTransitionExecutionOrder(transString));
        trans.setPriority(prio);
        lli = new LabelLayoutInformation();
        if ((mip[1] >= sep[1]) && (mip[0] >= sep[0])) {
            lli.setUpperLeftPoint(new Point(
                    sep[0] - parentPos[0] + 10,
                    sep[1] - parentPos[1] + 5));
        } else if ((mip[1] >= sep[1]) && (mip[0] < sep[0])) {
            lli.setUpperLeftPoint(new Point(
                    sep[0] - parentPos[0] - 10,
                    sep[1] - parentPos[1] + 5));
        } else if ((mip[1] < sep[1]) && (mip[0] >= sep[0])) {
            lli.setUpperLeftPoint(new Point(
                    sep[0] - parentPos[0] + 10,
                    sep[1] - parentPos[1] - 5));
        } else {
            lli.setUpperLeftPoint(new Point(
                    sep[0] - parentPos[0] - 10,
                    sep[1] - parentPos[1] - 5));
        }
        actualView.setLayoutInformation(prio, lli);
        EdgeLayoutInformation eli = new EdgeLayoutInformation();
        eli.addPathElement(new MovetoPath(sep[0] - parentPos[0],
                       sep[1] - parentPos[1]));
        //eli.addPathElement(new LinetoPath(mip[0] - parentPos[0],
        //                mip[1] - parentPos[1]));
        //eli.addPathElement(new LinetoPath(dep[0], dep[1]));
        eli.addPathElement(new QuadraticCurvetoPath(
                      mip[0] - parentPos[0],
                      mip[1] - parentPos[1],
                      dep[0], dep[1]));
        actualView.setLayoutInformation(trans, eli);

        //eli.addPathElement(new MovetoPath(
        //         mip[0] - parentPos[0],
        //         mip[1] - parentPos[1]));
        //eli.addPathElement(new MovetoPath(dep[0], dep[1]));
    }
}
```

```
/**
 * Grabs the complete Stateflow chart.
 * @return the grabbed Stateflow chart as <code>StateChart</code>
 * @throws IOException if an error occurs in the input or output stream.
 * @throws MatlabStateflowGrabberException if an error is encountered
 * by Matlab.
 */
public static StateChart grabStateflow() throws IOException,
    MatlabStateflowGrabberException {
    fromMatlab = MatlabProcessInfo.fromMatlabStream;
    toMatlab = MatlabProcessInfo.toMatlabStream;
    matlabPrompt = MatlabStateflowProperties.getMatlabPromptString();
    matlabErrorIndicator =
        MatlabStateflowProperties.getMatlabErrorIndicatorString();
    containsSubcharts = false;
    chartBoundary[0] = 0;
    chartBoundary[1] = 0;

    if (log == null) {
        try {
            log = new LogFile(new FileWriter(
                    System.getProperty("user.home") + File.separator
                    + ".kiel" + File.separator
                    + "MatlabStateflowGrabber.log"),
                    "MatlabStateflowGrabber");
        } catch (IOException e) {
            log = new LogFile(new PrintWriter(System.out),
                    "MatlabStateflowGrabber");
            log.log(0, "Could␣not␣open␣log␣file!");
        }
        log.setLogLevel(0);
        log.enableLog();
        log.setModuleName("MatlabStateflowGrabber");
    }

    log.log(0, "Grabbing␣Stateflow−Chart:");
    StateChart stc = new StateChart();
    stc.setModelSource("Matlab/Stateflow");
    stc.setModelVersion("6.1");
    actualView = new View();
    map.clear();

    ORState parent = new ORState();
    //parent.setName("root");
    stc.setRootNode(parent);
    CompositeStateLayoutInformation rootlayout =
        new CompositeStateLayoutInformation();
    rootlayout.setUpperLeftPoint(new Point(0, 0));

    /* Get Statechart object out of Matlab*/
    toMatlab.write(GET_ROOT_COMMAND);
    toMatlab.newLine();
    toMatlab.flush();
    checkMatlabResponse();
    toMatlab.write(GET_MODELS_COMMAND);
    toMatlab.newLine();
    toMatlab.flush();
    checkMatlabResponse();
    toMatlab.write(GET_CHART_COMMAND);
    toMatlab.newLine();
    toMatlab.flush();
    checkMatlabResponse();
    currentChartName = getObjectName("chart");
    parent.setName(currentChartName);
    map.put(getObjectID("chart"), parent);
    //int[] chartSize = getChartSize();
    //rootlayout.setWidth(chartSize[0]);
    //rootlayout.setHeight(chartSize[1]);
    //actualView.setLayoutInformation(parent, rootlayout);

    /*Get all Events and Variables
     * which are children of Stateflow.Chart
     * */
    log.log(0, "Grab␣input,␣output␣and␣local␣events:");
    toMatlab.write("inputEvents=chart" + GET_INPUTEVENTS_COMMAND);
    toMatlab.newLine();
    toMatlab.flush();
    checkMatlabResponse();
    toMatlab.write("outputEvents=chart" + GET_OUTPUTEVENTS_COMMAND);
    toMatlab.newLine();
    toMatlab.flush();
    checkMatlabResponse();
    toMatlab.write("localEvents=chart" + GET_LOCALEVENTS_COMMAND);
    toMatlab.newLine();
    toMatlab.flush();
    checkMatlabResponse();

    int noe = getArraySizeOf("inputEvents");
    String eventString;
    String trigger;
    int port;
    Event[] events = new Event[noe];

    for (int eventNumber = 0; eventNumber < noe; eventNumber++) {
        eventString = "inputEvents(" + (eventNumber + 1) + ")";
        port = getPortNumber(eventString);
        trigger = getTriggerType(eventString);
        events[eventNumber] = new IntegerSignal(
                getObjectName(eventString),
                new IntegerConstant(0), trigger, port);
    }
    stc.setInputEvents(events);

    noe = getArraySizeOf("outputEvents");
    events = new Event[noe];

    for (int eventNumber = 0; eventNumber < noe; eventNumber++) {
        eventString = "outputEvents(" + (eventNumber + 1) + ")";
         port = getPortNumber(eventString);
         trigger = getTriggerType(eventString);
         events[eventNumber] = new IntegerSignal(
                getObjectName(eventString),
                new IntegerConstant(0), trigger, port);
    }
    stc.setOutputEvents(events);

    noe = getArraySizeOf("localEvents");
    events = new Event[noe];
```

```
        for (int eventNumber = 0; eventNumber < noe; eventNumber++) {
            eventString = "localEvents(" + (eventNumber + 1) + ")";
             events[eventNumber] = new Event(
                    getObjectName(eventString));
        }
        stc.setLocalEvents(events);

        ArrayList eventsCol = new ArrayList();
        for (int i = 0; i < events.length; i++) {
            eventsCol.add(events[i]);
        }
        stc.getRootNode().addLocalEvents(eventsCol);

        getAndCreateVariables(stc);

        /* Get all states out of Matlab level by level of containment*/
        toMatlab.write("parent_states=chart" + GET_STATES_COMMAND);
        toMatlab.newLine();
        toMatlab.flush();
        checkMatlabResponse();
        toMatlab.write("parent=chart");
        toMatlab.newLine();
        toMatlab.flush();
        checkMatlabResponse();
        grabOneLevel();

        int nos;
        String stateString;
        boolean first = true;

        nos = getArraySizeOf("parent_states");

        do {
            toMatlab.write("nextLevelParent_states=[]");
            toMatlab.newLine();
            toMatlab.flush();
            checkMatlabResponse();

            for (int stateNumber = 0; stateNumber < nos; stateNumber++) {
                stateString = "parent_states(" + (stateNumber + 1) + ")";
                if (!isSimpleState(stateString)) {
                    toMatlab.write("parent=" + stateString);
                    toMatlab.newLine();
                    toMatlab.flush();
                    checkMatlabResponse();
                    grabOneLevel();
                    if (first) {
                        toMatlab.write("nextLevelParent_states=states");
                        toMatlab.newLine();
                        toMatlab.flush();
                        checkMatlabResponse();
                        first = false;
                    } else {
                        toMatlab.write("nextLevelParent_states="
                                + "cat(1,␣nextLevelParent_states,␣states)");
                        toMatlab.newLine();
                        toMatlab.flush();
                        checkMatlabResponse();
                    }
                }
            }

            toMatlab.write("parent_states=nextLevelParent_states");
            toMatlab.newLine();
            toMatlab.flush();
            checkMatlabResponse();
            first = true;
        } while ((nos = getArraySizeOf("nextLevelParent_states")) > 0);

        rootlayout.setWidth(chartBoundary[0] + 15);
        rootlayout.setHeight(chartBoundary[1] + 15);
        actualView.setLayoutInformation(parent, rootlayout);

        //map.clear(); We need the map in the simulator!!!
        closeLogFile();
        return stc;
    }

    /**
     * Returns the previously constructed view. If the grabbed chart contains
     * subcharts, then no view will be returned.
     * @return - the view that results from importing the chart, null if the
     * chart contains subcharts.
     */
    public static View getView() {
        if (containsSubcharts) {
            return null;
        } else {
            return actualView;
        }
    }

    /**
     * Closes the log file.
     */
    public static void closeLogFile() {
        if (log != null) {
            log.closeLog();
            log = null;
        }
    }
}
```

B.5.2. MatlabStateflowGrabberException

```java
//$Id: MatlabStateflowGrabberException.java,v 1.1 2005/06/30 16:22:03 apo Exp $
package kiel.util;

/**
 * @author apo
 * @version $Revision: 1.1 $ last modified $Date: 2005/06/30 16:22:03 $
 */
public class MatlabStateflowGrabberException extends Exception {
        public MatlabStateflowGrabberException(final String message) {
        super(message);
    }
}
```

B.5.3. MatlabStateflowCreator

```java
//$Id:
package kiel.util;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.Collection;
import java.util.HashMap;
import java.util.Iterator;

import kiel.dataStructure.ANDState;
import kiel.dataStructure.CompositeState;
import kiel.dataStructure.Constant;
import kiel.dataStructure.History;
import kiel.dataStructure.InitialState;
import kiel.dataStructure.Junction;
import kiel.dataStructure.Node;
import kiel.dataStructure.Region;
import kiel.dataStructure.State;
import kiel.dataStructure.StateChart;
import kiel.dataStructure.StringLabel;
import kiel.dataStructure.StringVariable;
import kiel.dataStructure.Transition;
import kiel.dataStructure.Variable;
import kiel.dataStructure.action.Actions;
import kiel.dataStructure.boolexp.BooleanFalse;
import kiel.dataStructure.boolexp.BooleanTrue;
import kiel.dataStructure.boolexp.BooleanVariable;
import kiel.dataStructure.doubleexp.DoubleConstant;
import kiel.dataStructure.doubleexp.DoubleVariable;
import kiel.dataStructure.eventexp.Event;
import kiel.dataStructure.eventexp.IntegerSignal;
import kiel.dataStructure.floatexp.FloatConstant;
import kiel.dataStructure.floatexp.FloatVariable;
import kiel.dataStructure.int16exp.Integer16Constant;
import kiel.dataStructure.int16exp.Integer16Variable;
import kiel.dataStructure.int8exp.Integer8Constant;
import kiel.dataStructure.int8exp.Integer8Variable;
import kiel.dataStructure.intexp.IntegerConstant;
import kiel.dataStructure.intexp.IntegerVariable;
import kiel.dataStructure.uint16exp.UInt16Constant;
import kiel.dataStructure.uint16exp.UInt16Variable;
import kiel.dataStructure.uint32exp.UInt32Constant;
import kiel.dataStructure.uint32exp.UInt32Variable;
import kiel.dataStructure.uint8exp.UInt8Constant;
import kiel.dataStructure.uint8exp.UInt8Variable;
import kiel.graphicalInformations.EdgeLayoutInformation;
import kiel.graphicalInformations.LabelLayoutInformation;
import kiel.graphicalInformations.NodeLayoutInformation;
import kiel.graphicalInformations.PathElement;
import kiel.graphicalInformations.Point;
import kiel.graphicalInformations.View;

/**
 * @author apo
 * @version $Revision: 1.8 $ last modified $Date: 2005/12/15 13:31:12 $
 */
public final class MatlabStateflowCreator {
    /**
     * The Matlab prompt, usually '>> '.
     */
    private static String matlabPrompt;

    /**
     * The string that indicates an error in Matlab, usually '???'.
     */
    private static String matlabErrorIndicator;

    /**
     * The buffered reader used to read Matlab's output.
     */
    private static BufferedReader fromMatlab;

    /**
     * The buffered writer used to send commands to Matlab.
     */
    private static BufferedWriter toMatlab;

    /**
     * The view which includes the graphical information (layout) of the
     * currently processed chart.
     */
    private static View actualView;

    /**
     * The last answer of Matlab.
     */
    private static String input = "";

    /**
     * The file the log is written into.
     */
    private static LogFile log = null;

    /**
     * The currently processed chart.
     */
    private static StateChart chart;

    /**
     * The has map that is used to map between Stateflow objects and
     * KIEL objects.
     */
    private static HashMap map = new HashMap();

    /**
     * The command to close all Simulink models.
     */
```

```java
private static final String CLOSE_OPEN_CHARTS_COMMAND = "sfexit";

/**
 * The command to create a new Simulink model with one Stateflow chart.
 */
private static final String CREATE_NEW_CHART_COMMAND = "sfnew";

/**
 * The command to obtain the Stateflow root object.
 */
private static final String GET_ROOT_COMMAND = "rt=sfroot";

/**
 * The command to obtain the Simulink model.
 */
private static final String GET_MODELS_COMMAND =
    "model=rt.find('-isa','Simulink.BlockDiagram')";

/**
 * The command to obtain the Stateflow chart contained in the model.
 */
private static final String GET_CHART_COMMAND =
    "chart=model.find('-isa','Stateflow.Chart')";

/**
 * The suffix to access the type of a variable.
 */
private static final String DTYPE = ".DataType";

/**
 * The suffix to access the unique identifier of a Stateflow object.
 */
private static final String ID = ".Id";

/**
 * The suffix to access the name of a Stateflow object.
 */
private static final String NAME = ".Name";

/**
 * The suffix to access the label of a state of transition.
 */
private static final String LABEL = ".LabelString";

/**
 * The suffix to access the position of a transition label.
 */
private static final String LABEL_POS = ".LabelPosition";

/**
 * The suffix to access the position of a graphical Stateflow object.
 */
private static final String POSITION = ".Position";

/**
 * The suffix to access the center cooridinates of a junction.
 */
private static final String JUNC_POS = ".Position.Center";

/**
 * The suffix to access the radius of a junction.
 */
private static final String JUNC_RADIUS = ".Position.Radius";

/**
 * The suffix to access the midpoint of a transition.
 */
private static final String TR_MIDPOINT = ".MidPoint";

/**
 * The suffix to access the position and size of
 * the window that contains the chart.
 */
private static final String CHART_SIZE = ".WindowPosition";

/**
 * The suffix to access the source of a transition.
 */
private static final String SRC = ".Source";

/**
 * The suffix to access the source end point of a transition.
 */
private static final String SRC_EP = ".SourceEndPoint";

/**
 * The suffix to access the destination of a transition.
 */
private static final String DEST = ".Destination";

/**
 * The suffix to access the o'clock position of a transition source.
 */
private static final String SRC_OCLOCK = ".SourceOClock";

/**
 * The suffix to access the o'clock position of a transition destination.
 */
private static final String DEST_OCLOCK = ".DestinationOClock";

/**
 * The suffix to access the decomposition type of a state.
 */
private static final String DECOMPO = ".Decomposition";

/**
 * The suffix to access the type of a transition (connective, history).
 */
private static final String TYPE = ".Type";

/**
 * The suffix to optain the port of an input or output event.
 */
private static final String PORT = ".Port";

/**
 * The suffix to access the trigger type of an input event.
 */
private static final String TRIGGER = ".Trigger";
```

```
/**
 * The suffix to access the scope of a variable (local, input, or output).
 */
private static final String SCOPE = ".Scope";

/**
 * The suffix to access the initial value of a variable.
 */
private static final String INITIAL_VALUE = ".props.InitialValue";

/**
 * The constructor for states.
 */
private static final String CREATE_STATE = "Stateflow.State(";

/**
 * The constructor for transitions.
 */
private static final String CREATE_TRANS = "Stateflow.Transition(";

/**
 * The constructor for junctions.
 */
private static final String CREATE_JUNC = "Stateflow.Junction(";

/**
 * The constructor for events.
 */
private static final String CREATE_EVENT = "Stateflow.Event(";

/**
 * The constructor for variables (data).
 */
private static final String CREATE_VARIABLE = "Stateflow.Data(";

/**
 * The suffix to check whether a state intersects with
 * one ore more other states.
 */
private static final String BAD_INTERSECTION = ".BadIntersection";

/**
 * Size of the input buffer (for receiving Matlab's answer).
 */
private static final int BUFF_SIZE = 20;

/**
 * Utility classes should not have a public or default constructor.
 */
private MatlabStateflowCreator() { }

/**
 * Receives output from Matlab. This method uses a buffered reader
 * that is connected to Matlab's standard output. The stream is read
 * until a Matlab prompt is encountered. Matlab's answer is stored
 * in the global variable <code>input</code>. If an error is indicated
 * by Matlab, then an exception is thrown.
 * @throws IOException if an error occurs in the input or output stream.
 * @throws MatlabStateflowCreatorException if an error is encountered
 * by Matlab.
 */
private static void checkMatlabResponse() throws IOException,
    MatlabStateflowCreatorException {
    input = "";
    char[] inputBuf = new char[BUFF_SIZE];
    int numRead = 0;

    while (!input.endsWith(matlabPrompt)) {
        numRead = fromMatlab.read(inputBuf, 0, inputBuf.length);
        if (numRead > 0) {
            input += new String(inputBuf, 0, numRead);
        }
    }
    log.log(0, input);
    if (input.indexOf(matlabErrorIndicator) != -1) {
        closeLogFile();
        throw new MatlabStateflowCreatorException(input);
    }
}

/**
 * Sends a command to Matlab and receives Matlab's response. This method
 * calls <code>checkMatlabResponse</code> to receive Matlab's answer to
 * the command.
 * @param command - the command that has to be sent to Matlab.
 * @return Matlab's answer to the command.
 * @throws IOException if an error occurs in the input or output stream.
 * @throws MatlabStateflowCreatorException if an error is encountered
 * by Matlab.
 */
private static String[] handleMatlabCommand(final String command)
    throws IOException, MatlabStateflowCreatorException {
    log.log(0, "Handle␣Matlab␣command:␣" + command);
    toMatlab.write(command);
    toMatlab.newLine();
    toMatlab.flush();
    checkMatlabResponse();
    return input.trim().split("\\s+");
}

/**
 * Closes the log file.
 */
public static void closeLogFile() {
    if (log != null) {
        log.closeLog();
        log = null;
    }
}

/**
 * Obtains the position of the object specified by
 * the argument <code>obj</code>.
 * @param obj - the object for which one wants to know the position.
 * @return the coordinates of the object [x, y ,width, height].
 * @throws IOException if an error occurs in the input or output stream.
 * @throws MatlabStateflowCreatorException if an error is encountered
 * by Matlab.
 */
private static int[] getObjectPosition(final String obj)
```

```java
    throws IOException, MatlabStateflowCreatorException {
    String[] answer = handleMatlabCommand(obj + POSITION);
    int[] pos = new int[4];
    pos[0] = (int) Float.parseFloat(answer[2]);
    pos[1] = (int) Float.parseFloat(answer[3]);
    pos[2] = (int) Float.parseFloat(answer[4]);
    pos[3] = (int) Float.parseFloat(answer[5]);

    return pos;
}

/**
 * Obtains the port number of an event or data object.
 * @param eventOrData - the event or data object for which one wants to
 * know the port number.
 * @return the port number of the specified event or data.
 * @throws IOException if an error occurs in the input or output stream.
 * @throws MatlabStateflowCreatorException if an error is encountered
 * by Matlab.
 */
private static int getPortNumber(final String eventOrData)
    throws IOException, MatlabStateflowCreatorException {
    String[] answer = handleMatlabCommand(eventOrData + PORT);
    return Integer.parseInt(answer[2]);
}

/**
 * Obtains the port number of all signals specified by the parameter
 * <code>col</code>. This method sets the <code>port</code> property
 * of the signals to the obtained values.
 * @param col - the collection of signals for which one wants to
 * know the port numbers.
 * @throws IOException if an error occurs in the input or output stream.
 * @throws MatlabStateflowCreatorException if an error is encountered
 * by Matlab.
 */
private static void getAllPortNumbers(final Collection col)
    throws IOException, MatlabStateflowCreatorException {
    Iterator it = col.iterator();
    IntegerSignal signal;

    while (it.hasNext()) {
        signal = (IntegerSignal) it.next();
        signal.setPort(getPortNumber(signal.getName()
                + "_" + signal.hashCode()));
    }
}

/**
 * Obtains the unique identifier of the Stateflow object
 * specified by <code>obj</code>.
 * @param obj - the object for which one wants to know the id.
 * @return the id of obj.
 * @throws IOException if an error occurs in the input or output stream.
 * @throws MatlabStateflowCreatorException if an error is encountered
 * by Matlab.
 */
private static Integer getObjectID(final String obj) throws IOException,
    MatlabStateflowCreatorException {
    String[] answer = handleMatlabCommand(obj + ID);
    return new Integer(answer[2]);
}

/**
 * Sets the name of the Stateflow object specified by
 * the <code>obj</code> argument to <code>name</code>.
 * @param obj - the object one wants to set the name of
 * @param name - the name of the object.
 * @throws IOException if an error occurs in the input or output stream.
 * @throws MatlabStateflowCreatorException if an error is encountered
 * by Matlab.
 */
private static void setObjectName(final String obj, final String name)
    throws IOException, MatlabStateflowCreatorException {
    String command = obj + NAME + "='" + name + "';";
    log.log(0, "Set␣object␣name:␣" + command);
    toMatlab.write(command);
    toMatlab.newLine();
    toMatlab.flush();
    checkMatlabResponse();
}

/**
 * Sets the label of a state or of a transition specified by
 * the <code>obj</code> argument to <code>label</code>.
 * @param obj - the state or transition one wants to set the label of.
 * @param label - the label of obj.
 * @throws IOException if an error occurs in the input or output stream.
 * @throws MatlabStateflowCreatorException if an error is encountered
 * by Matlab.
 */
private static void setObjectLabel(final String obj, final String label)
    throws IOException, MatlabStateflowCreatorException {
    String command = obj + LABEL + "=sprintf('" + label + "');";
    log.log(0, "Set␣object␣label:␣" + command);
    toMatlab.write(command);
    toMatlab.newLine();
    toMatlab.flush();
    checkMatlabResponse();
}

/**
 * Sets the position of the label of the object specified by
 * the <code>obj</code> argument to <code>pos</code>.
 * @param obj - the object (state or transition) of which one
 * wants to set the label position [x, y, width, height].
 * @param pos - the coordinates of the label of the object obj.
 * @throws IOException if an error occurs in the input or output stream.
 * @throws MatlabStateflowCreatorException if an error is encountered
 * by Matlab.
 */
private static void setObjectLabelPosition(final String obj,
                                           final int [] pos)
    throws IOException, MatlabStateflowCreatorException {
    String command = obj + LABEL_POS
        + "=[" + pos[0] + "␣"
        + pos[1] + "␣" + pos[2] + "␣" + pos[3] + "];";
    log.log(0, "Set␣object␣label␣position:␣" + command);
    toMatlab.write(command);
    toMatlab.newLine();
```

```java
        toMatlab.flush();
        checkMatlabResponse();
    }

    /**
     * Sets the position of the object specified by
     * the argument <code>obj</code> to <code>pos</code>.
     * @param obj - the object of which one wants to set the position.
     * @param pos - the coordinates of the object [x, y ,width, height].
     * @throws IOException if an error occurs in the input or output stream.
     * @throws MatlabStateflowCreatorException if an error is encountered
     * by Matlab.
     */
    private static void setObjectPosition(final String obj, final int[] pos)
        throws IOException, MatlabStateflowCreatorException {
        String command = obj + POSITION
            + "=[" + pos[0] + "␣"
            + pos[1] + "␣" + pos[2] + "␣" + pos[3] + "];";
        log.log(0, "Set␣object␣position:␣" + command);
        toMatlab.write(command);
        toMatlab.newLine();
        toMatlab.flush();
        checkMatlabResponse();
    }

    /**
     * Sets the scope of the variable or event specified by
     * the <code>obj</code> argument to <code>scope</code>.
     * @param obj - the variable one wants to set the scope of.
     * @param scope - the scope of the variable or event
     * (input, output, local).
     * @throws IOException if an error occurs in the input or output stream.
     * @throws MatlabStateflowCreatorException if an error is encountered
     * by Matlab.
     */
    private static void setObjectScope(final String obj, final String scope)
        throws IOException, MatlabStateflowCreatorException {
        String command = obj + SCOPE + "='" + scope + "';";
        log.log(0, "Set␣object␣scope:␣" + command);
        toMatlab.write(command);
        toMatlab.newLine();
        toMatlab.flush();
        checkMatlabResponse();
    }

    /**
     * Sets the source of the transition specified by <code>trans</code>
     * to <code>src</code>.
     * @param trans - the transition for which one wants to set the source.
     * @param src - the source of the transition <code>trans</code>.
     * @throws IOException if an error occurs in the input or output stream.
     * @throws MatlabStateflowCreatorException if an error is encountered
     * by Matlab.
     */
    private static void setTransitionSource(final String trans,
                                            final String src)
        throws IOException, MatlabStateflowCreatorException {
        String command = trans + SRC + "=" + src + ";";
        log.log(0, "Set␣transition␣source:␣" + command);
        toMatlab.write(command);
        toMatlab.newLine();
        toMatlab.flush();
        checkMatlabResponse();
    }

    /**
     * Sets the destination of the transition specified
     * by <code>trans</code> to <code>dest</code>.
     * @param trans - the transition of which one wants to set
     * the destination.
     * @param dest - the destination of the transition <code>trans</code>.
     * @throws IOException if an error occurs in the input or output stream.
     * @throws MatlabStateflowCreatorException if an error is encountered
     * by Matlab.
     */
    private static void setTransitionDestination(final String trans,
                                                 final String dest)
        throws IOException, MatlabStateflowCreatorException {
        String command = trans + DEST + "=" + dest + ";";
        log.log(0, "Set␣transition␣destination:␣" + command);
        toMatlab.write(command);
        toMatlab.newLine();
        toMatlab.flush();
        checkMatlabResponse();
    }

    /**
     * Calculates and sets the o'clock value according to the source end
     * point of the transition specified by <code>trans</code> and
     * <code>sep</code>.
     * @param trans - the transition of which one wants to set
     * the source end point.
     * @param sep - the source end point [x, y].
     * @param srcState - the state that is the source of the transition.
     * @param parentPos - the position of the parent state of <code>srcState
     * </code>.
     * @throws IOException if an error occurs in the input or output stream.
     * @throws MatlabStateflowCreatorException if an error is encountered
     * by Matlab.
     */
    private static void setTransitionSourceEndPoint(final String trans,
                                                    final int[] sep,
                                                    final Node srcState,
                                                    final int[] parentPos)
        throws IOException, MatlabStateflowCreatorException {
        NodeLayoutInformation nli = actualView.getLayoutInformation(srcState);
        int[] center = {parentPos[0] + nli.getXPos() + nli.getWidth() / 2,
                parentPos[1] + nli.getYPos() + nli.getHeight() / 2};
        double c = Math.sqrt((double) ((sep[0] - center[0])
                * (sep[0] - center[0])
                + (sep[1] - center[1]) * (sep[1] - center[1])));
        double oclockRad = Math.atan2(sep[1] - center[1], sep[0] - center[0]);
        oclockRad += Math.PI / 2;
        if (oclockRad < 0) {
            oclockRad = (2 * Math.PI) + oclockRad;
        }
        double oclock = oclockRad / (2 * Math.PI / 12);

        String command = trans + SRC_OCLOCK + "=" + oclock + ";";
        log.log(0, "Set␣transition␣source␣end␣point:␣" + trans + SRC_EP + "="
                + "(" + sep[0] + ",␣" + sep[1]
                + ")␣This␣corresponds␣to␣oclock:␣" + oclock
                + "\nDestination␣center:␣("
                + center[0] + ",␣" + center[1] + ")"
                + "\ncommand:␣" + command);
        toMatlab.write(command);
        toMatlab.newLine();
        toMatlab.flush();
        checkMatlabResponse();
    }

    /**
     * Sets the transition source end point directly without calculating
     * the o'clock value.
     * @param trans - the transition of which one wants to set
     * the source end point.
     * @param sep - the source end point [x, y].
     * @throws IOException if an error occurs in the input or output stream.
     * @throws MatlabStateflowCreatorException if an error is encountered
     * by Matlab.
     */
    private static void setTransitionSourceEndPointDirect(final String trans,
                                                          final int[] sep)
        throws IOException, MatlabStateflowCreatorException {
        String command = trans + SRC_EP + "=[" + sep[0] + "␣" + sep[1] + "];";
        log.log(0, "Set␣transition␣source␣end␣point:␣" + command);
        toMatlab.write(command);
        toMatlab.newLine();
        toMatlab.flush();
        checkMatlabResponse();
    }

    /**
     * Sets the midpoint of the transition specified by <code>trans</code>.
     * @param trans - the transition of which one wants to set the midpoint.
     * @param mip - the midpoint [x, y].
     * @throws IOException if an error occurs in the input or output stream.
     * @throws MatlabStateflowCreatorException if an error is encountered
     * by Matlab.
     */
    private static void setTransitionMidPoint(final String trans,
                                              final int[] mip)
        throws IOException, MatlabStateflowCreatorException {
        String command = trans + TR_MIDPOINT + "="
            + "[" + mip[0] + "␣" + mip[1] + "];";
        log.log(0, "Set␣transition␣mid␣point:␣" + command);
        toMatlab.write(command);
        toMatlab.newLine();
        toMatlab.flush();
        checkMatlabResponse();
    }

    /**
     * Calculates and sets the destination o'clock value of the transition
     * specified by <code>trans</code>.
     * @param trans - the transition for which one wants to know the
     * destination end point.
     * @param destState - the state that is the destination of the transition
     * specified by <code>trans</code>.
     * @param dep - the destination end point [x, y].
     * @param parentPos - the position of the parent state of <code>destState
     * </code>.
     * @throws IOException if an error occurs in the input or output stream.
     * @throws MatlabStateflowCreatorException if an error is encountered
     * by Matlab.
     */
    private static void setTransitionDestinationEndPoint(final String trans,
                                                         final int[] dep,
                                                         final Node destState,
                                                         final int[] parentPos)
        throws IOException, MatlabStateflowCreatorException {
        NodeLayoutInformation nli = actualView.getLayoutInformation(destState);
        int[] center = {parentPos[0] + nli.getXPos() + nli.getWidth() / 2,
                parentPos[1] + nli.getYPos() + nli.getHeight() / 2};
        double c = Math.sqrt((double) ((dep[0] - center[0])
                * (dep[0] - center[0])
                + (dep[1] - center[1]) * (dep[1] - center[1])));
        double oclockRad = Math.atan2(dep[1] - center[1], dep[0] - center[0]);
        oclockRad += Math.PI / 2;
        if (oclockRad < 0) {
            oclockRad = (2 * Math.PI) + oclockRad;
        }
        double oclock = oclockRad / (2 * Math.PI / 12);

        String command = trans + DEST_OCLOCK + "=" + oclock + ";";
        log.log(0, "Set␣transition␣destination␣end␣point:␣(" + dep[0]
                + ",␣" + dep[1] + ")␣This␣corresponds␣to␣oclock:␣"
                + oclock + "\nDestination␣center:␣("
                + center[0] + ",␣" + center[1] + ")"
                + "\ncommand:␣" + command);
        toMatlab.write(command);
        toMatlab.newLine();
        toMatlab.flush();
        checkMatlabResponse();
    }

    /**
     * Sets the position of the junction specified by the argument
     * <code>junction</code> to <code>pos</code>.
     * @param junc - the junction for which one wants to know the position.
     * @param pos - the position of the junction specified by the argument.
     * @throws IOException if an error occurs in the input or output stream.
     * @throws MatlabStateflowCreatorException if an error is encountered
     * by Matlab.
     */
    private static void setJunctionPosition(final String junc, final int[] pos)
        throws IOException, MatlabStateflowCreatorException {
        String command = junc + JUNC_POS + "=[" + pos[0] + "␣" + pos[1] + "];";
        log.log(0, "Set␣junction␣position:␣" + command);
        toMatlab.write(command);
        toMatlab.newLine();
        toMatlab.flush();
        checkMatlabResponse();
    }

    /**
     * Sets the radius of the junction specified by the argument
     * <code>junction</code> to <code>rad</code>.
     * @param junc - the junction for which one wants to know the radius.
     * @param rad - the radius of the specified junction.
```

```
 * @throws IOException if an error occurs in the input or output stream.
 * @throws MatlabStateflowCreatorException if an error is encountered
 * by Matlab.
 */
private static void setJunctionRadius(final String junc, final int rad)
    throws IOException, MatlabStateflowCreatorException {
    String command = junc + JUNC_RADIUS + "=" + rad + ";";
    log.log(0, "Set␣junction␣radius:␣" + command);
    toMatlab.write(command);
    toMatlab.newLine();
    toMatlab.flush();
    checkMatlabResponse();
}

/**
 * Sets the initial value of the variable specified by the argument
 * <code>obj</code> to <code>val</code>.
 * @param obj - the variable of which one wants to set the initial value.
 * @param val - the initial value of the variable <code>obj</code>.
 * @throws IOException if an error occurs in the input or output stream.
 * @throws MatlabStateflowCreatorException if an error is encountered
 * by Matlab.
 */
private static void setInitialValue(final String obj, final String val)
    throws IOException, MatlabStateflowCreatorException {
    String command = obj + INITIAL_VALUE + "='" + val + "';";
    log.log(0, "Set␣initial␣value␣of␣variable:␣" + command);
    toMatlab.write(command);
    toMatlab.newLine();
    toMatlab.flush();
    checkMatlabResponse();
}

/**
* Sets the trigger type of the event specified by the argument
* <code>event</code> to <code>type</code>.
* @param event - the event of which one wants to know the trigger type.
* @param type the trigger type of the specified event
* (rising, falling, either, function).
* @throws IOException if an error occurs in the input or output stream.
* @throws MatlabStateflowCreatorException if an error is encountered
* by Matlab.
*/
private static void setTriggerType(final String event, final String type)
    throws IOException, MatlabStateflowCreatorException {
    String command = event + TRIGGER + "='" + type + "';";
    log.log(0, "Set␣trigger␣type␣of␣signal:␣" + command);
    toMatlab.write(command);
    toMatlab.newLine();
    toMatlab.flush();
    checkMatlabResponse();
}

/**
 * Sets the data type of the variable specified by
 * the <code>data</code> argument to <code>type</code>.
 * @param data - the variable one wants to know the data type of
 * @param type - the data type of the variable (int32, int16, int8,
 * single, double, boolean, unsigned integer (32, 16, 8 bit).
 * @throws IOException if an error occurs in the input or output stream.
 * @throws MatlabStateflowCreatorException if an error is encountered
 * by Matlab.
 */
private static void setDataType(final String data, final String type)
    throws IOException, MatlabStateflowCreatorException {
    String command = data + DTYPE + "='" + type + "';";
    log.log(0, "Set␣data␣type:␣" + command);
    toMatlab.write(command);
    toMatlab.newLine();
    toMatlab.flush();
    checkMatlabResponse();
}

/**
 * Creates a new object in Stateflow.
 * @param obj - the handle which holds the created object.
 * @param parent - the handle of the parent that contains the object.
 * @param type - the type of object to create (transition, state, etc.).
 * @throws IOException if an error occurs in the input or output stream.
 * @throws MatlabStateflowCreatorException if an error is encountered
 * by Matlab.
 */
private static void createObject(final String obj,
                                 final String parent,
                                 final String type)
    throws IOException, MatlabStateflowCreatorException {
    String command = obj + "=" + type + parent + ");";
    log.log(0, "Create␣new␣object:␣" + command);
    toMatlab.write(command);
    toMatlab.newLine();
    toMatlab.flush();
    checkMatlabResponse();
}

/**
 * Creates the local variables in Stateflow contained in the state
 * specified by the argument <code>state</code>.
 * @param state - the state that contains the variables to be created
 * in Stateflow.
 * @throws IOException if an error occurs in the input or output stream.
 * @throws MatlabStateflowCreatorException if an error is encountered
 * by Matlab.
 */
private static void getAndCreateLocalVariables(final State state)
    throws IOException, MatlabStateflowCreatorException {
    Collection col = state.getVariables();
    createVariables(col, state, "Local");
    col = state.getConstants();
    createConstants(col, state);
}

/**
 * Checks whether a state in Stateflow intersects with other states
 * by inspecting the property <code>BadIntersection</code> of that state.
 * @param state - the state that has to be checked.
 * @return true if a bad intersection persists, false otherwise.
 * @throws IOException if an error occurs in the input or output stream.
 * @throws MatlabStateflowCreatorException if an error is encountered
 * by Matlab.
 */
```

```java
    private static boolean hasBadIntersection(final String state)
        throws IOException, MatlabStateflowCreatorException {
        String[] answer = handleMatlabCommand(state + BAD_INTERSECTION);
        return (answer[2].equals("1"));
    }

    /**
     * Creates the variables in Stateflow that are specified by the parameter
     * <code>col</code>.
     * @param col - the variables that have to be created.
     * @param parent - the handle of the parent state that
     * is expected to contain the variables.
     * @param scope - the scope of the variables (local, input, output).
     * @throws IOException if an error occurs in the input or output stream.
     * @throws MatlabStateflowCreatorException if an error is encountered
     * by Matlab.
     */
    private static void createVariables(final Collection col,
                                        final State parent,
                                        final String scope)
        throws IOException, MatlabStateflowCreatorException {
        Object item;
        Variable variable;
        String name;
        String identifier;
        String parentIdentifier = parent.getName() + "_" + parent.hashCode();;
        Iterator it = col.iterator();

        while (it.hasNext()) {
            item = it.next();
            variable = (Variable) item;
            name = variable.getName();
            identifier = name + "_" + variable.hashCode();
            createObject(identifier, parentIdentifier, CREATE_VARIABLE);
            setObjectName(identifier, name);

            if (item.getClass().equals(IntegerVariable.class)) {
                setDataType(identifier, "int32");
            } else if (item.getClass().equals(Integer16Variable.class)) {
                setDataType(identifier, "int16");
            } else if (item.getClass().equals(Integer8Variable.class)) {
                setDataType(identifier, "int8");
            } else if (item.getClass().equals(UInt32Variable.class)) {
                setDataType(identifier, "uint32");
            } else if (item.getClass().equals(UInt16Variable.class)) {
                setDataType(identifier, "uint16");
            } else if (item.getClass().equals(UInt8Variable.class)) {
                setDataType(identifier, "uint8");
            } else if (item.getClass().equals(FloatVariable.class)) {
                setDataType(identifier, "single");
            } else if (item.getClass().equals(DoubleVariable.class)) {
                setDataType(identifier, "double");
            } else if (item.getClass().equals(BooleanVariable.class)) {
                setDataType(identifier, "boolean");
                if (variable.toExpString().equals("true")) {
                    setInitialValue(identifier, "1");
                } else {
                    setInitialValue(identifier, "0");
                }
                setObjectScope(identifier, scope);
                continue;
            } else if (item.getClass().equals(StringVariable.class)) {
                throw new MatlabStateflowCreatorException(
                        "Chart␣contains␣variable␣of␣unsupported␣type!␣"
                        + "Cannot␣create␣variable␣of␣type␣'String'.");
            } else {
                throw new MatlabStateflowCreatorException(
                        "Chart␣contains␣variable␣of␣unsupported␣type:␣"
                        + item.getClass());
            }
            setObjectScope(identifier, scope);
            setInitialValue(identifier, variable.toExpString());
        }
    }

    /**
     * Creates the constants in Stateflow that are specified by the parameter
     * <code>col</code>.
     * @param col - the constants that have to be created.
     * @param parent - the handle of the parent state that
     * is expected to contain the constants.
     * @throws IOException if an error occurs in the input or output stream.
     * @throws MatlabStateflowCreatorException if an error is encountered
     * by Matlab.
     */
    private static void createConstants(final Collection col,
                                        final State parent)
        throws IOException, MatlabStateflowCreatorException {
        Object item;
        Constant constant;
        String name;
        String identifier;
        String parentIdentifier = parent.getName() + "_" + parent.hashCode();;
        Iterator it = col.iterator();

        while (it.hasNext()) {
            item = it.next();
            constant = (Constant) item;
            name = constant.getName();
            identifier = name + "_" + constant.hashCode();
            createObject(identifier, parentIdentifier, CREATE_VARIABLE);
            setObjectName(identifier, name);

            if (item.getClass().equals(IntegerConstant.class)) {
                setDataType(identifier, "int32");
            } else if (item.getClass().equals(Integer16Constant.class)) {
                setDataType(identifier, "int16");
            } else if (item.getClass().equals(Integer8Constant.class)) {
                setDataType(identifier, "int8");
            } else if (item.getClass().equals(UInt32Constant.class)) {
                setDataType(identifier, "uint32");
            } else if (item.getClass().equals(UInt16Constant.class)) {
                setDataType(identifier, "uint16");
            } else if (item.getClass().equals(UInt8Constant.class)) {
                setDataType(identifier, "uint8");
            } else if (item.getClass().equals(FloatConstant.class)) {
                setDataType(identifier, "single");
            } else if (item.getClass().equals(DoubleConstant.class)) {
                setDataType(identifier, "double");
            } else if (item.getClass().equals(BooleanTrue.class)) {
                setDataType(identifier, "boolean");
                setInitialValue(identifier, "1");
            } else if (item.getClass().equals(BooleanFalse.class)) {
                setDataType(identifier, "0");
                setInitialValue(identifier, "1");
                setObjectScope(identifier, "Contant");
                continue;
            } else {
                throw new MatlabStateflowCreatorException(
                        "Chart␣contains␣constant␣of␣unsupported␣type:␣"
                        + item.getClass());
            }
            setObjectScope(identifier, "Constant");
            setInitialValue(identifier, constant.toExpString());
        }
    }

    /**
     * Creates the local events in Stateflow contained in the state
     * specified by the argument <code>state</code>.
     * @param parent - the state that contains the events to be created
     * in Stateflow.
     * @throws IOException if an error occurs in the input or output stream.
     * @throws MatlabStateflowCreatorException if an error is encountered
     * by Matlab.
     */
    private static void getAndCreateLocalEvents(final State parent)
        throws IOException, MatlabStateflowCreatorException {
        Event event;
        String name;
        String identifier;
        Collection col = parent.getLocalEvents();
        Iterator it = col.iterator();

        while (it.hasNext()) {
            event = (Event) it.next();
            name = event.getName();
            identifier = name + "_" + event.hashCode();
            createObject(identifier, "chart", CREATE_EVENT);
            setObjectName(identifier, name);
        }
    }

    /**
     * Returns all incoming and outgoing transitions of the state in KIEL
     * specified by the argument <code>parent</code>.
     * @param parent - the state in KIEL which one wants to know
     * the transitions of.
     * @return the incoming and outgoing transitions of the state.
     */
    private static ArrayList getTransitions(final State parent) {
        if (!(parent instanceof CompositeState)) {
            return new ArrayList();
        }
        ArrayList children = ((CompositeState) parent).getSubnodes();
        ArrayList transitions = new ArrayList();
        Node node;
        Object item;

        Iterator it = children.iterator();
        while (it.hasNext()) {
            item = it.next();
            if (item instanceof Node) {
                node = (Node) item;
                transitions.addAll(node.getOutgoingTransitions());
            }
        }

        return transitions;
    }

    /**
     * Decreases the size of the state <code>state</code> as long as a bad
     * intersection for this state persists.
     * @param state - the handle of the state whose bad intersection has to
     * be resolved.
     * @throws IOException if an error occurs in the input or output stream.
     * @throws MatlabStateflowCreatorException if an error is encountered
     * by Matlab or if the bad intersection cannot be resolved.
     */
    private static void resolveBadIntersection(final String state)
        throws IOException, MatlabStateflowCreatorException {
        while (hasBadIntersection(state)) {
            int[] pos = getObjectPosition(state);
            pos[0] += 1;
            pos[1] += 1;
            pos[2] -= 2;
            pos[3] -= 2;
            if (pos[2] < 5) {
                throw new MatlabStateflowCreatorException(
                        "Cannot␣resolve␣bad␣intersection!");
            }
            setObjectPosition(state, pos);
        }
    }

    /**
     * Creates all objects in Stateflow (child states, transitions, variables)
     * that are contained in the state <code>parent</code>.
     * @param parent - the handle of the state whose child objects have to
     * be created.
     * @return all children of <code>parent</code>.
     * @throws IOException if an error occurs in the input or output stream.
     * @throws MatlabStateflowCreatorException if an error is encountered
     * by Matlab.
     */
    private static ArrayList createOneLevel(final State parent)
        throws IOException, MatlabStateflowCreatorException {
        ArrayList states = new ArrayList();
        ArrayList transitions;
        ArrayList junctions = new ArrayList();
        ArrayList nodes;
        Node node;
        NodeLayoutInformation nli;
        EdgeLayoutInformation eli;
        LabelLayoutInformation lli;
        String name;
        String label;
        String identifier;
        String parentIdentifier;
```

```java
String sourceIdentifier;
String destinationIdentifier;
State state;
Transition trans;
Node sourceNode;
Node destinationNode;
Transition transition;
Actions actions;

if (parent.getName().length() == 0) {
    parentIdentifier = "state_" + parent.hashCode();
} else {
    parentIdentifier = parent.getName() + "_" + parent.hashCode();
}
nli = actualView.getLayoutInformation(parent);
int[] parentPos = null;
if (chart.getRootNode() != parent) {
    parentPos = getObjectPosition(parentIdentifier);
} else {
    parentPos = new int[4];
}

if (parent instanceof CompositeState) {
    nodes = ((CompositeState) parent).getSubnodes();

    Iterator it = nodes.iterator();
    while (it.hasNext()) {
        node = (Node) it.next();
        if (node instanceof State) {
            states.add(node);
        } else if (!(node instanceof InitialState)) {
            junctions.add(node);
        }
    }
}

int nos = states.size();
int[] pos = new int[4];

for (int stateNumber = 0; stateNumber < nos; stateNumber++) {
    state = (State) states.get(stateNumber);
    name = state.getName();
    if (name.length() == 0) {
        identifier = "state_" + state.hashCode();
    } else {
        identifier = name + "_" + state.hashCode();
    }
    createObject(identifier, parentIdentifier, CREATE_STATE);
    map.put(getObjectID(identifier), state);

    nli = actualView.getLayoutInformation(
            (State) states.get(stateNumber));
    pos[0] = nli.getXPos() + parentPos[0];
    pos[1] = nli.getYPos() + parentPos[1];
    pos[2] = nli.getWidth();
    pos[3] = nli.getHeight();
    setObjectPosition(identifier, pos);
    //setObjectName(identifier, name);
    label = name;
    actions = state.getEntry();
    if (actions.toString().length() > 0) {
        label += "\\nentry:␣" + actions.toString() + ";";
    }

    actions = state.getDoActivity();
    if (actions.toString().length() > 0) {
        label += "\\nduring:␣" + actions.toString() + ";";
    }

    actions = state.getExit();
    if (actions.toString().length() > 0) {
        label += "\\nexit:␣" + actions.toString() + ";";
    }

    actions = state.getBindAction();
    if (actions.toString().length() > 0) {
        label += "\\nbind:␣" + actions.toString() + ";";
    }

    trans = state.getInternalTransition();
    if (((StringLabel) trans.getLabel()).toString().length() > 0) {
        label += "\\n" + ((StringLabel) trans.getLabel()).toString()
            + ";";
    }

    setObjectLabel(identifier, label);

    if (state instanceof ANDState) {
        handleMatlabCommand(identifier + DECOMPO + "='PARALLEL_AND';");
    }
    if (state instanceof Region) {
        resolveBadIntersection(identifier);
    }
    getAndCreateLocalEvents(state);
    getAndCreateLocalVariables(state);
}

int nojs = junctions.size();
pos = new int[2];
Object item;

for (int i = 0; i < nojs; i++) {
    item = junctions.get(i);
    nli = actualView.getLayoutInformation((Node) item);
    pos[0] = nli.getXPos() + parentPos[0] + nli.getWidth();
    pos[1] = nli.getYPos() + parentPos[1] + nli.getWidth();
    if (item instanceof History) {
        createObject("hist_" + item.hashCode(),
                parentIdentifier, CREATE_JUNC);
        handleMatlabCommand("hist_" + item.hashCode()
                + TYPE + "='HISTORY';");
        setJunctionPosition("hist_" + item.hashCode(), pos);
        setJunctionRadius("hist_" + item.hashCode(),
                nli.getWidth() / 2);
    } else {
        createObject("junc_" + item.hashCode(),
                parentIdentifier, CREATE_JUNC);
        handleMatlabCommand("junc_" + item.hashCode()
                + TYPE + "='CONNECTIVE';");
        setJunctionPosition("junc_" + item.hashCode(), pos);
```

```
                setJunctionRadius("junc_" + item.hashCode(),
                        nli.getWidth() / 2);
            }
        }

        transitions = getTransitions(parent);
        int nots = transitions.size();
        ArrayList path;
        Point point;
        int[] sep = new int[2];
        int[] dep = new int[2];
        int[] mip = new int[2];

        for (int transitionNumber = 0; transitionNumber < nots;
            transitionNumber++) {
            transition = (Transition) transitions.get(transitionNumber);
            name = ((StringLabel) (transition.getLabel())).toString();
            name = name.replaceAll("\n", "\\\\n");
            identifier = "trans_" + transition.hashCode();
            createObject(identifier, parentIdentifier, CREATE_TRANS);
            setObjectLabel(identifier, name);
            sourceNode = (Node) transition.getSource();
            if (sourceNode instanceof Junction) {
                sourceIdentifier = "junc_" + sourceNode.getName()
                    + sourceNode.hashCode();
            } else {
                if (sourceNode.getName().length() == 0) {
                    sourceIdentifier = "state_" + sourceNode.hashCode();
                } else {
                    sourceIdentifier = sourceNode.getName() + "_"
                        + sourceNode.hashCode();
                }
            }
            destinationNode = (Node) transition.getTarget();
            if (destinationNode instanceof Junction) {
                destinationIdentifier = "junc_" + destinationNode.getName()
                    + destinationNode.hashCode();
            } else {
                if (destinationNode.getName().length() == 0) {
                    destinationIdentifier = "state_"
                        + destinationNode.hashCode();
                } else {
                    destinationIdentifier = destinationNode.getName() + "_"
                        + destinationNode.hashCode();
                }
            }

            eli = actualView.getLayoutInformation(transition);
            path = eli.getAllPathElements();
            point = ((PathElement) path.get(0)).getTargetPoint();
            sep[0] = point.getX() + parentPos[0];
            sep[1] = point.getY() + parentPos[1];

            if (!(sourceNode instanceof InitialState)) {
                setTransitionSource(identifier, sourceIdentifier);
                setTransitionSourceEndPoint(identifier,
                        sep,
                        sourceNode,
                        parentPos);
            } else {
                setTransitionSourceEndPointDirect(identifier, sep);
            }
            setTransitionDestination(identifier, destinationIdentifier);

            point = ((PathElement) path.get(path.size() - 1)).getTargetPoint();
            dep[0] = point.getX() + parentPos[0];
            dep[1] = point.getY() + parentPos[1];
            setTransitionDestinationEndPoint(identifier,
                    dep,
                    destinationNode,
                    parentPos);
            point = ((PathElement) path.get(path.size() / 2)).getTargetPoint();
            mip[0] = point.getX() + parentPos[0];
            mip[1] = point.getY() + parentPos[1];
            //setTransitionMidPoint(identifier, mip);

            lli = actualView.getLayoutInformation(transition.getLabel());
            if (lli != null) {
                point = lli.getUpperLeft();
                pos = new int[4];
                pos[0] = point.getX() + parentPos[0];
                pos[1] = point.getY() + parentPos[1];
                pos[2] = lli.getWidth() + 10;
                pos[3] = lli.getHeight() + 10;
                setObjectLabelPosition(identifier, pos);
            }
        }
        // do this for children already!
        //getAndCreateLocalEvents(parent);
        //getAndCreateLocalVariables(parent);
        return states;
    }

    /**
     * Creates a complete chart in Stateflow.
     * @param stc - the chart that has to be created in Stateflow.
     * @param view - the view that belongs to the chart.
     * @throws IOException if an error occurs in the input or output stream.
     * @throws MatlabStateflowCreatorException if an error is encountered
     * by Matlab.
     */
    public static void createChart(final StateChart stc, final View view)
        throws IOException, MatlabStateflowCreatorException {
        if (MatlabProcessInfo.matlabProcess == null) {
            try {
                MatlabProcessInfo.startMatlab();
            } catch (IOException ex) {

            }
        }
        fromMatlab = MatlabProcessInfo.fromMatlabStream;
        toMatlab = MatlabProcessInfo.toMatlabStream;
        matlabPrompt = MatlabStateflowProperties.getMatlabPromptString();
        matlabErrorIndicator =
            MatlabStateflowProperties.getMatlabErrorIndicatorString();
        actualView = view;
        chart = stc;
        map.clear();

        if (log == null) {
```

```
    try {
        log = new LogFile(new FileWriter(
                System.getProperty("user.home") + File.separator
                + ".kiel" + File.separator
                + "MatlabStateflowCreator.log"),
                "MatlabStateflowCreator");
    } catch (IOException e) {
        log = new LogFile(new PrintWriter(System.out),
                "MatlabStateflowCreator");
        log.log(0, "Could␣not␣open␣log␣file!");
    }
    log.setLogLevel(0);
    log.enableLog();
    log.setModuleName("MatlabStateflowCreator");
}

log.log(0, "Creating␣Stateflow−Chart:");

/*Create new chart in Matlab, close all open ones before*/
toMatlab.write(CLOSE_OPEN_CHARTS_COMMAND);
toMatlab.newLine();
toMatlab.flush();
checkMatlabResponse();
toMatlab.write(CREATE_NEW_CHART_COMMAND);
toMatlab.newLine();
toMatlab.flush();
checkMatlabResponse();

toMatlab.write(GET_ROOT_COMMAND);
toMatlab.newLine();
toMatlab.flush();
checkMatlabResponse();
toMatlab.write(GET_MODELS_COMMAND);
toMatlab.newLine();
toMatlab.flush();
checkMatlabResponse();
toMatlab.write(GET_CHART_COMMAND);
toMatlab.newLine();
toMatlab.flush();
checkMatlabResponse();

/*Create local variables and events*/
Collection col;
Iterator it;
IntegerSignal event;
Variable variable;
Object item;
String name;
String identifier;

/*Create input events*/
col = stc.getInputEvents();
it = col.iterator();

while (it.hasNext()) {
    event = (IntegerSignal) it.next();
    name = event.getName();
    identifier = name + "_" + event.hashCode();
    createObject(identifier, "chart", CREATE_EVENT);
    setObjectName(identifier, name);
    setObjectScope(identifier, "Input");
    setTriggerType(identifier, event.getTrigger());
}

/*Create output events*/
col = stc.getOutputEvents();
it = col.iterator();

while (it.hasNext()) {
    event = (IntegerSignal) it.next();
    name = event.getName();
    identifier = name + "_" + event.hashCode();
    createObject(identifier, "chart", CREATE_EVENT);
    setObjectName(identifier, name);
    setObjectScope(identifier, "Output");
    setTriggerType(identifier, event.getTrigger());
}

/*Temporarily change root name*/
String chartName = stc.getRootNode().getName();
toMatlab.write("RootNode_"
        + stc.getRootNode().hashCode() + "=chart;");
toMatlab.newLine();
toMatlab.flush();
checkMatlabResponse();

stc.getRootNode().setName("RootNode");

/*Create input variables*/
col = stc.getInputVariables();
createVariables(col, stc.getRootNode(), "Input");

/*Create output variables*/
col = stc.getOutputVariables();
createVariables(col, stc.getRootNode(), "Output");

/*Create local variables*/
col = stc.getLocalVariables();
createVariables(col, stc.getRootNode(), "Local");

/*Create constants*/
col = stc.getConstants();
createConstants(col, stc.getRootNode());

/*Get all port numbers. Simulink assigns port numbers to
 * all input and output signals while they are created.*/
getAllPortNumbers(stc.getInputEvents());
getAllPortNumbers(stc.getOutputEvents());

/*Create chart level by level of containment*/
ArrayList parentStates = createOneLevel(stc.getRootNode());
ArrayList nextLevelParentStates = new ArrayList();
int nos = parentStates.size();
stc.getRootNode().setName(chartName);

do {
    nextLevelParentStates.clear();

    for (int i = 0; i < nos; i++) {
        nextLevelParentStates
```

```
                .addAll(createOneLevel((State) parentStates.get(i)));
        }
        parentStates = new ArrayList(nextLevelParentStates);
    } while ((nos = nextLevelParentStates.size()) > 0);
```

B.5.4. MatlabStateflowCreatorException

```
//$Id:
package kiel.util;

/**
 * @author apo
 * @version $Revision: 1.1 $ last modified $Date: 2005/09/16 10:18:51 $
 */
```

```
        MatlabStateflowGrabber.map = map;
    }
}
```

```
public class MatlabStateflowCreatorException extends Exception {
    public MatlabStateflowCreatorException(final String message) {
        super(message);
    }
}
```

B.5.5. MatlabProcessInfo

```java
//$Id: MatlabProcessInfo.java,v 1.7 2005/12/15 13:31:12 apo Exp $
package kiel.util;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;

/**
 * @author apo
 * @version $Revision: 1.7 $ last modified $Date: 2005/12/15 13:31:12 $
 */
public final class MatlabProcessInfo {
    /**
     * The name of the Matlab executalbe file name.
     */
    private static String matlabExecutableFileName;

    /**
     * Parameters for the MatlabExecutable.
     */
    private static String options = "";

    /**
     * The Matlab prompt, usually '>> '.
     */
    private static String matlabPrompt;

    /**
     * Holds a reference to the Matlab process.
     */
    public static Process matlabProcess;

    /**
     * Holds a reference to the output stream of Matlab.
     */
    public static BufferedReader fromMatlabStream;

    /**
     * Holds a reference to the input stream of Matlab.
     */
    public static BufferedWriter toMatlabStream;

    /**
     * Size of the input buffer.
     */
    private static final int BUFF_SIZE = 20;

    /**
     * Utility classes should not have a public or default constructor.
     */
    private MatlabProcessInfo() { }

    /**
     * If and only if an instance of Matlab had been started by KIEL
     * then this method does the following:
     * First it tells Matlab to quit and waits until Matlab has terminated.
     * If that does not work it uses Process.destroy();
     */
    public static void shutdownMatlab() {
        if (matlabProcess != null) {
            try {
                toMatlabStream.write("sfexit");
                toMatlabStream.newLine();
                toMatlabStream.flush();
                fromMatlabStream.skip(3);
                toMatlabStream.write("quit");
                toMatlabStream.newLine();
                toMatlabStream.flush();

                try {
                    matlabProcess.waitFor();
                    toMatlabStream.close();
                    fromMatlabStream.close();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            } catch (IOException e) {
                matlabProcess.destroy();
            } finally {
                matlabProcess = null;
                fromMatlabStream = null;
                toMatlabStream = null;
            }
        }
    }

    /**
     * If no instance of Matlab is running, this method starts a new one.
     * @return Matlab's response.
     * @throws IOException - if something goes wrong while communicating with
     * Matlab.
     */
    public static String startMatlab() throws IOException {
        String input = "";
        char[] inputBuf = new char[BUFF_SIZE];
        int numRead = 0;
        MatlabStateflowProperties.loadProps();
        matlabExecutableFileName =
            MatlabStateflowProperties.getMatlabExecutableFileName();
        matlabPrompt = MatlabStateflowProperties.getMatlabPromptString();
        if (MatlabStateflowProperties.startMatlabWithNodisplay()) {
            options = "␣-nodisplay";
        } else {
            options = "";
        }

        if (matlabProcess == null) {
            matlabProcess =
                Runtime.getRuntime().exec(matlabExecutableFileName + options);
            fromMatlabStream =
                new BufferedReader(new InputStreamReader(
                        matlabProcess.getInputStream()));
            toMatlabStream =
                new BufferedWriter(new OutputStreamWriter(
                    matlabProcess.getOutputStream()));

            while (input.indexOf(matlabPrompt) == -1) {
                numRead = fromMatlabStream.read(
                        inputBuf, 0, inputBuf.length);
                input += new String(inputBuf, 0, numRead);
            }
            String returnValue = new String(input);

            /*This is a workaround due to a bug in Matlab which causes
             * Matlab not to show a prompt after command has
             * finished its output.
             */
            toMatlabStream.write("open_system('bla')");
            toMatlabStream.newLine();
            //enter a new line to force Matlab to show a prompt
            toMatlabStream.newLine();
            toMatlabStream.flush();

            input = "";
            while (input.indexOf(matlabPrompt) == -1) {
                numRead = fromMatlabStream.read(
                        inputBuf, 0, inputBuf.length);
                input += new String(inputBuf, 0, numRead);
            }
            return returnValue;
        }
        return new String();
    }
}
```

B.5.6. MatlabStateflowProperties

```
//$Id:
package kiel.util;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.FileWriter;
import java.io.IOException;
import java.io.InputStream;

/**
 * @author apo
 * @version $Revision: 1.3 $ last modified $Date: 2005/10/06 14:52:35 $
 */

public class MatlabStateflowProperties {

    /**
     * The property class containing the values.
     */
    private static final java.util.Properties INTERNAL_PROPERTIES = new java.util
     .
        Properties();

    /**
     * path to user properties.
     */
    private static final String FILE = System.getProperty("user.home")
        + File.separator + ".kiel" + File.separator
        + "matlabStateflow.properties";

    /**
     * Not for use.
     */
    private MatlabStateflowProperties() {
    }

    /**
     * Loads user properties file or creates default properties
     * if no user file avaliable.
     */
    public static void loadProps() {
        try {
                INTERNAL_PROPERTIES.load(new FileInputStream(new File(FILE)));
        } catch (IOException ex1) {
                createDefaultFile();
            loadProps();
        }
    }

    private static void createDefaultFile() {
        InputStream is;
        FileWriter fw;

        try {
            is = MatlabStateflowProperties.class.getClassLoader().
                getResourceAsStream(
                    "kiel/simulator/matlab/matlabStateflow.properties");
            fw = new FileWriter(FILE);

            int c = is.read();
            while (c >= 0) {
              fw.write(c);
              c = is.read();
            }
            fw.flush();

            is.close();
            fw.close();
        } catch (IOException storeException) {

        }
    }

    public static boolean startMatlabWithNodisplay() {
        String answer = INTERNAL_PROPERTIES.getProperty(
                "MatlabProcessInfo.nodisplay", "false");
        return answer.equals("true");
    }

    public static float getChartScale() {
        String answer = INTERNAL_PROPERTIES.getProperty(
                "MatlabStateflowGrabber.chartScale", "1.0");
        try {
                return Float.parseFloat(answer);
        } catch (NumberFormatException e) {
                return 1.0f;
        }
    }

    public static String getInputVariableNamePrefix() {
        return INTERNAL_PROPERTIES.getProperty(
                "StateflowSimulator.inputVariableNamePrefix", "VarMat");
    }

    public static String getMatlabPromptString() {
        return INTERNAL_PROPERTIES.getProperty(
                "matlabPrompt", ">>␣");
    }

    public static String getMatlabErrorIndicatorString() {
        return INTERNAL_PROPERTIES.getProperty(
                "matlabErrorIndicator", "???");
    }

    public static String getMatlabExecutableFileName() {
        return INTERNAL_PROPERTIES.getProperty(
                "MatlabProcessInfo.matlabExecutableFileName", "matlab");
    }
}
```

B.6. kiel.simulator.matlab

B.6.1. StateflowSimulator

```java
//$Id: StateflowSimulator.java,v 1.22 2005/12/15 16:14:48 apo Exp $
package kiel.simulator.matlab;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;

import kiel.configMngr.Configuration;
import kiel.configMngr.MacroStep;
import kiel.configMngr.SignalAbsent;
import kiel.configMngr.SignalPresent;
import kiel.configMngr.StateActivated;
import kiel.configMngr.StateDeactivated;
import kiel.configMngr.VariableValue;
import kiel.dataStructure.CompositeState;
import kiel.dataStructure.Node;
import kiel.dataStructure.State;
import kiel.dataStructure.StateChart;
import kiel.dataStructure.Variable;
import kiel.dataStructure.eventexp.Event;
import kiel.dataStructure.eventexp.IntegerSignal;
import kiel.simulator.Simulator;
import kiel.simulator.SimulatorException;
import kiel.util.Benchmark;
import kiel.util.LogFile;
import kiel.util.MatlabProcessInfo;
import kiel.util.MatlabStateflowGrabber;
import kiel.util.MatlabStateflowProperties;

/**
 * @author apo
 * @version $Revision: 1.22 $ last modified $Date: 2005/12/15 16:14:48 $
 */
public class StateflowSimulator implements Simulator {

    /**
     * The Matlab prompt, usually '>> '.
     */
    private static String matlabPrompt;

    /**
     * The string that indicates an error in Matlab, usually '???'.
     */
    private static String matlabErrorIndicator;

    /**
     * The prefix used for variables in the Simulink environment.
     */
    private static String inputVariableNamePrefix;

    /**
     * The chart that is currently simulated.
     */
    private StateChart stc;

    /**
     * The buffered reader used to read Matlab's output.
     */
    private BufferedReader fromMatlab;

    /**
     * The buffered writer used to send commands to Matlab.
     */
    private BufferedWriter toMatlab;

    /**
     * The last answer of Matlab.
     */
    private String input = "";

    /**
     * The file name of the currently processed chart.
     */
    private String fileName = "";

    /**
     * The name of the currently processed chart.
     */
    private String chartName = "";

    /**
     * The file the log is written into.
     */
    private LogFile log;

    /**
     * This array associates states with output ports. The state whose
     * port number is i is stored in the array at position (i - 1).
     */
    private State[] stateActivityOutput;

    /**
     * This array holds all input signals.
     */
    private IntegerSignal[] inputSignals;

    /**
     * This array holds all output signals.
     */
    private IntegerSignal[] outputSignals;
```

```
/**
 * This array associates states and output signals with output ports.
 * The state or output signal whose port number is i is stored in the array
 * at position (i - 1).
 */
private Object[] simoutArray;

/**
 * This array stores the actual values of all input signals. The signale
 * whose port number is i is stored at position (i - 1).
 */
private int[] signalValues;

/**
 * This array stores the actual values of all input variables.
 */
private double[] inputVarsValues;

/**
 * This array stores all variable names.
 */
private String[] variableNames;

/**
 * This array stores the previous values of the variables.
 */
private Object[] prevVariableValues;

/**
 * This array contains all variables of the chart.
 */
private Object[] variables;

/**
 * This array contains all input variables.
 */
private Object[] inputVariables;

/**
 * This array stores the strings used to access the input events in the
 * Matlab workspace.
 */
private String[] inputEventsStrings;

/**
 * This flag is true if the triggers have been modified by the simulator
 * to support the event simulation mode.
 */
private boolean triggerModified = false;

/**
 * The number of output ports (of the chart).
 */
private int noops;

/**
 * The size of the matrix that is the output of the 'simout' block. This
 * matrix contains the simulation results except for output variables.
 */
private int simsize;

/**
 * The number of the current simulation step.
 */
private int currentStepNumber = 0;

/**
 * Size of the input buffer (for receiving Matlab's answer).
 */
private static final int BUFF_SIZE = 20;
private Benchmark simBench;

/**
 * Creates a new simulator object. It connects the simulator to Matlab's
 * output and input streams.
 */
public StateflowSimulator() {
    if (MatlabProcessInfo.matlabProcess == null) {
        try {
            MatlabProcessInfo.startMatlab();
        } catch (IOException ex) {

        }
    }

    fromMatlab = MatlabProcessInfo.fromMatlabStream;
    toMatlab = MatlabProcessInfo.toMatlabStream;
    matlabPrompt = MatlabStateflowProperties.getMatlabPromptString();
    matlabErrorIndicator =
        MatlabStateflowProperties.getMatlabErrorIndicatorString();
    inputVariableNamePrefix =
        MatlabStateflowProperties.getInputVariableNamePrefix();

    simBench = new Benchmark();
    simBench.setModuleName("StateflowSimulator");
    simBench.setPreMessage("Messuring␣sim␣step␣time");
}

/**
 * Receives output from Matlab. This method uses a buffered reader
 * that is connected to Matlab's standard output. The stream is read
 * until a Matlab prompt is encountered. Matlab's answer is stored
 * in the global variable <code>input</code>. If an error is indicated
 * by Matlab, then an exception is thrown.
 * @throws IOException if an error occurs in the input or output stream.
 * @throws SimulatorException if an error is encountered
 * by Matlab.
 */
private void checkMatlabResponse() throws IOException, SimulatorException {
    input = "";
    char[] inputBuf = new char[BUFF_SIZE];
    int numRead = 0;

    while (!input.endsWith(matlabPrompt)) {
        numRead = fromMatlab.read(inputBuf, 0, inputBuf.length);
        if (numRead > 0) {
            input += new String(inputBuf, 0, numRead);
        }
    }
```

```
        log.log(0, input);
        if (input.indexOf(matlabErrorIndicator) != -1) {
            throw new SimulatorException(input);
        }
    }

    /**
     * Sends a command to Matlab and receives Matlab's response. This method
     * calls <code>checkMatlabResponse</code> to receive Matlab's answer to
     * the command.
     * @param command - the command that has to be sent to Matlab.
     * @return Matlab's answer to the command.
     * @throws IOException if an error occurs in the input or output stream.
     * @throws SimulatorException if an error is encountered
     * by Matlab.
     */
    private String[] handleMatlabCommand(final String command)
        throws IOException, SimulatorException {
        log.log(0, "Handle␣Matlab␣command:␣" + command);
        toMatlab.write(command);
        toMatlab.newLine();
        toMatlab.flush();
        checkMatlabResponse();
        return input.trim().split("\\s+");
    }

    /**
     * Obtains the position of the chart block in the Simulink diagram.
     * @return the position (and size) of the chart block in the Simulink
     * diagram [left, top, right, bottom].
     * @throws IOException if an error occurs in the input or output stream.
     * @throws SimulatorException if an error is encountered
     * by Matlab.
     */
    private int[] getChartPosition() throws IOException, SimulatorException {
        String[] answer = handleMatlabCommand("get_param('"
                + fileName
                + "/" + chartName
                + "','Position')");
        int[] pos = new int[4];
        pos[0] = (int) Float.parseFloat(answer[2]); //left
        pos[1] = (int) Float.parseFloat(answer[3]); //top
        pos[2] = (int) Float.parseFloat(answer[4]); //right
        pos[3] = (int) Float.parseFloat(answer[5]); //bottom
        return pos;
    }

    /**
     * Obtains the number of output ports of the chart.
     * @return The number of output ports of the chart.
     * @throws IOException if an error occurs in the input or output stream.
     * @throws SimulatorException if an error is encountered
     * by Matlab.
     */
    private int getNumberOfOutputPorts() throws IOException,
        SimulatorException {
        String[] answer = handleMatlabCommand("get_param('"
                + fileName
                + "/" + chartName
                + "','Ports')");
        return Integer.parseInt(answer[3]);
    }

    /**
     * Obtains the port number of the event or data specified by the parameter
     * <code>eventOrData</code>.
     * @param eventOrData - the handle of the event or data of which one
     * wants to know the port number.
     * @return the port number of the event or data.
     * @throws IOException if an error occurs in the input or output stream.
     * @throws SimulatorException if an error is encountered
     * by Matlab.
     */
    private int getPortNumber(final String eventOrData)
        throws IOException, SimulatorException {
        String[] answer = handleMatlabCommand(eventOrData + ".Port");
        return Integer.parseInt(answer[2]);
    }

    /**
     * Sets the trigger of each event to 'Either'.
     * @throws IOException if an error occurs in the input or output stream.
     * @throws SimulatorException if an error is encountered
     * by Matlab.
     */
    private void setTriggerToEither() throws IOException,
        SimulatorException {
        int port;
        for (int i = 0; i < inputSignals.length; i++) {
            if (!(inputSignals[i].getTrigger()).equals("Either")) {
                port = inputSignals[i].getPort();
                handleMatlabCommand(inputEventsStrings[port - 1]
                    + ".Trigger='Either'");
            }
        }
        triggerModified = true;
    }

    /**
     * Restores the trigger setting of each event to its original setting.
     * @throws IOException if an error occurs in the input or output stream.
     * @throws SimulatorException if an error is encountered
     * by Matlab.
     */
    private void restoreTrigger() throws IOException,
        SimulatorException {
        int port;
        for (int i = 0; i < inputSignals.length; i++) {
            if (!(inputSignals[i].getTrigger()).equals("Either")) {
                port = inputSignals[i].getPort();
                handleMatlabCommand(inputEventsStrings[port - 1] + ".Trigger="
                        + "'" + inputSignals[i].getTrigger() + "'");
            }
        }
        triggerModified = false;
    }

    /**
     * Creates an array that contains all state objects. Each state is stored
     * at the position (i - 1) where i is its port number.
```

```java
     * @throws IOException if an error occurs in the input or output stream.
     * @throws SimulatorException if an error is encountered
     * by Matlab.
     */
    private void createStateActivityOutputArray() throws IOException,
        SimulatorException {
        int nos;
        String[] answer;
        Integer id;
        String stateString;

        handleMatlabCommand("StateActivityOutput=chart.find('-isa',"
                + "'Stateflow.Data','-depth',1,'Scope','Output',"
                + "'DataType','State')");
        answer = handleMatlabCommand("size(StateActivityOutput)");
        nos = Integer.parseInt(answer[2]);
        stateActivityOutput = new State[noops];

        for (int i = 1; i <= nos; i++) {
            stateString = "StateActivityOutput(" + i + ")";
            answer = handleMatlabCommand(stateString + ".OutputState.Id");
            id = new Integer(answer[2]);
            answer = handleMatlabCommand(stateString + ".Port");
            stateActivityOutput[Integer.parseInt(answer[2]) - 1] =
                (State) MatlabStateflowGrabber.map.get(id);
        }
    }

    /**
     * Creates the arrays <code>inputSignals</code>, <code>signalValues</code>
     * and <code>inputEventStrings</code>.
     * @throws IOException if an error occurs in the input or output stream.
     * @throws SimulatorException if an error is encountered
     * by Matlab.
     */
    private void createInputSignalsArray() throws IOException,
        SimulatorException {
        ArrayList inputSignalsArray = stc.getInputEvents();
        int port;
        String eventString;
        inputSignals = new IntegerSignal[inputSignalsArray.size()];
        signalValues = new int[inputSignals.length];

        for (int i = 0; i < inputSignalsArray.size(); i++) {
            port = ((IntegerSignal) inputSignalsArray.get(i)).getPort();
            inputSignals[port - 1] = (IntegerSignal) inputSignalsArray.get(i);

            signalValues[port - 1] = Integer.parseInt(
                    inputSignals[port - 1].getInitialValue()
                    .toIntExpString());
        }

        handleMatlabCommand("inputEvents=chart.find('-isa','Stateflow.Event',
    "
                + "'-depth',1,'Scope','Input')");
        inputEventsStrings = new String[inputSignalsArray.size()];
        for (int i = 0; i < inputSignalsArray.size(); i++) {
            eventString = "inputEvents(" + (i + 1) + ")";
            port = getPortNumber(eventString);
            inputEventsStrings[port - 1] = eventString;
        }

    }

    /**
     * Creates the array <code>outputSignals</code>. This array contains all
     * output signals. Each signal is stored at position (i - 1) where i is
     * the signals port number.
     * @throws IOException if an error occurs in the input or output stream.
     * @throws SimulatorException if an error is encountered
     * by Matlab.
     */
    private void createOutputSignalsArray() throws IOException,
        SimulatorException {
        ArrayList outputSignalsArray = stc.getOutputEvents();
        int port;
        outputSignals = new IntegerSignal[outputSignalsArray.size()];

        for (int i = 0; i < outputSignalsArray.size(); i++) {
            port = ((IntegerSignal) outputSignalsArray.get(i)).getPort();
            outputSignals[port - 1] =
                (IntegerSignal) outputSignalsArray.get(i);
        }
    }

    /**
     * Creates the array <code>simoutArray</code>. This array stores states and
     * output signals. The state or output signal whose port number is i is
     * stored in the array at position (i - 1).
     * @throws IOException if an error occurs in the input or output stream.
     * @throws SimulatorException if an error is encountered
     * by Matlab.
     */
    private void createSimoutArray() throws IOException,
        SimulatorException {
        int nos;
        String[] answer;
        Integer id;
        String stateString;
        simoutArray = new Object[noops];
        ArrayList outputSignalsArray = stc.getOutputEvents();
        int port;

        handleMatlabCommand("StateActivityOutput=chart.find('-isa',"
                + "'Stateflow.Data','-depth',1,'Scope','Output',"
                + "'DataType','State')");
        answer = handleMatlabCommand("size(StateActivityOutput)");
        nos = Integer.parseInt(answer[2]);

        for (int i = 1; i <= nos; i++) {
            stateString = "StateActivityOutput(" + i + ")";
            answer = handleMatlabCommand(stateString + ".OutputState.Id");
            id = new Integer(answer[2]);
            answer = handleMatlabCommand(stateString + ".Port");
            port = Integer.parseInt(answer[2]);
            simoutArray[port - 1] =
                (State) MatlabStateflowGrabber.map.get(id);
            handleMatlabCommand("add_line('" + fileName + "','"
                    + chartName + "/"
                    + port + "','route/" + port + "')");
        }

        for (int i = 0; i < outputSignalsArray.size(); i++) {
            port = ((IntegerSignal) outputSignalsArray.get(i)).getPort();
            port += nos;
            simoutArray[port - 1] = (IntegerSignal) outputSignalsArray.get(i);
            handleMatlabCommand("add_line('" + fileName + "','"
                    + chartName + "/"
                    + port + "','route/" + port + "')");
        }
        //simsize = nos + outputSignalsArray.size();
        simsize = noops;
    }

    /**
     * Enables the 'DataSaveToWorkspace' option of all data objects.
     * @throws IOException if an error occurs in the input or output stream.
     * @throws SimulatorException if an error is encountered
     * by Matlab.
     */
    private void enableDataSaveToWorkspace() throws IOException,
        SimulatorException {
        String dataString;
        String[] answer = handleMatlabCommand("data=chart.find('-isa',"
                + "'Stateflow.Data','-not','DataType','State',"
                + "'-not','Scope','Input')");
        answer = handleMatlabCommand("size(data)");
        int nod = Integer.parseInt(answer[2]);

        for (int i = 1; i <= nod; i++) {
            dataString = "data(" + i + ")";
            handleMatlabCommand(dataString + ".SaveToWorkspace=1");
            //answer = handleMatlabCommand(dataString + ".Name");
            //variableNames[i - 1] = answer[2];
        }
    }

    /**
     * Enables the 'HasOutputData' option of all states. States with this
     * option enabled output their status (aktivated or not) to Simulink.
     * @throws IOException if an error occurs in the input or output stream.
     * @throws SimulatorException if an error is encountered
     * by Matlab.
     */
    private void enableStateOutput() throws IOException,
        SimulatorException {
        String stateString;
        String[] answer = handleMatlabCommand("states=chart.find('-isa',"
                + "'Stateflow.State')");
        answer = handleMatlabCommand("size(states)");
        int nos = Integer.parseInt(answer[2]);

        for (int i = 1; i <= nos; i++) {
            stateString = "states(" + i + ")";
            handleMatlabCommand(stateString + ".HasOutputData=true");
        }
    }

    /**
     * Modifies the Simulink diagram in such a way that a simulation that
     * is controlled by the KIEL application is possible. 'From_Workspace'
     * and 'To_Workspace' blocks are created to provide a means to send input
     * to and recerive output from the Stateflow chart.
     * @throws IOException if an error occurs in the input or output stream.
     * @throws SimulatorException if an error is encountered
     * by Matlab.
     */
    private void createSimEnvForInputVariables() throws IOException,
        SimulatorException {
        String inString;
        String name;
        int port;
        int[] position; // [left top right bottom ]
        inputVariables = stc.getInputVariables().toArray();
        inputVarsValues = new double[inputVariables.length];
        handleMatlabCommand("in=chart.find('-isa','Stateflow.Data',"
                + "'-depth',1,'Scope','Input')");
        String[] answer = handleMatlabCommand("size(in)");
        int noi = Integer.parseInt(answer[2]);
        position = getChartPosition();

        for (int i = 1; i <= noi; i++) {
            inString = "in(" + i + ")";
            answer = handleMatlabCommand(inString + ".Port");
            port = Integer.parseInt(answer[2]);
            answer = handleMatlabCommand(inString + ".Name");
            name = answer[2];

            handleMatlabCommand("add_block('simulink/Sources/From␣Workspace','
"
                    + fileName + "/" + inString + "')");
            handleMatlabCommand("set_param('" + fileName
                    + "/" + inString + "','Position',["
                    + position[0] + "␣"
                    + (position[3] + 25 + i * 45) + "␣"
                    + (position[0] + 75) + "␣"
                    + (position[3] + 50 + i * 45) + "])");
            handleMatlabCommand("set_param('" + fileName
                    + "/" + inString + "','VariableName','" + name
                    + inputVariableNamePrefix + "')");
            handleMatlabCommand("add_block("
                    + "'simulink/Signal␣Attributes/Data␣Type␣Conversion','"
                    + fileName + "/" + "DTC" + i + "')");
            handleMatlabCommand("set_param('" + fileName
                    + "/" + "DTC" + i + "','Position',["
                    + (position[0] + 100) + "␣"
                    + (position[3] + 25 + i * 45) + "␣"
                    + (position[0] + 150) + "␣"
                    + (position[3] + 50 + i * 45) + "])");
            handleMatlabCommand("add_line('" + fileName + "','"
                    + inString + "/1','"
                    + "DTC" + i + "/1')");
            handleMatlabCommand("add_line('" + fileName + "','"
                    + "DTC" + i + "/1','"
                    + chartName + "/" + i + "')");
        }
    }

    /**
     * Collects all variables that are contained in the subchart that
     * starts at the state specified by the parameter <code>start</code>.
     * This method operates on the KIEL data structure only!
     * @param start - the state to start the collection of variables.
     * @return A collection of all variables that are contained starting
     * at the state <code>start</code>.
     */
    private Collection getAllVariables(final CompositeState start) {
        ArrayList list = new ArrayList();
        if (start.getVariables() != null && !start.getVariables().isEmpty()) {
          list.addAll(start.getVariables());
        }
        if (start.getSubnodes() != null && !start.getSubnodes().isEmpty()) {
          Iterator iter = start.getSubnodes().iterator();
          while (iter.hasNext()) {
            Node next = (Node) iter.next();
            if (next instanceof CompositeState) {
              list.addAll(getAllVariables((CompositeState) next));
            }
          }
        }
        return list;
    }

    /**
     * Creates the array <code>variables</code> that contains all variables
     * that exist in the chart.
     */
    private void createVariableArray() {
        Collection cvars = getAllVariables(stc.getRootNode());
        cvars.addAll(stc.getLocalVariables());
        cvars.addAll(stc.getOutputVariables());
        //if (cvars.size() > 0) {
            variables = cvars.toArray();
        //} else {
        //    variables = new Variable[0];
        //}
        variableNames = new String[variables.length];
        prevVariableValues = new Object[variables.length];
    }

    /**
     * Creates the arrays <code>prevVariableValues</code> and
     * <code>variableNames</code>.
     */
    private void createInitialPrevVariableValues() {
        for (int i = 0; i < variables.length; i++) {
            //if (variables[i] instanceof IntegerVariable) {
            //    prevVariableValues[i] = new Integer(
            //            (((IntegerVariable)variables[i])
            //                    .getInitialValue()).toIntExpString());
            //} else if (variables[i] instanceof DoubleVariable) {
            //    prevVariableValues[i] = new Double(
            //            (((DoubleVariable)variables[i])
            //                    .getInitialValue()).toDoubleExpString());
            //} else if (variables[i] instanceof FloatVariable) {
            //    prevVariableValues[i] = new Float(
            //            (((FloatVariable)variables[i])
            //                    .getInitialValue()).toFloatExpString());
            //} else if (variables[i] instanceof StringVariable) {
            //    prevVariableValues[i] = new String(
            //            ((StringVariable)variables[i]).toString());
            //}
            prevVariableValues[i] = ((Variable) variables[i]).toExpString();
            variableNames[i] = ((Variable) variables[i]).getName();
        }
    }

    /**
     * Sets the log file to the argument <code>newLog</code>.
     * @param newLog - specifies the log file.
     */
    public final void setLogFile(final LogFile newLog) {
        this.log = newLog;
    }

    /**
     * Deletes all blocks in the Simulink diagram specified by the argument
     * <code>blocks</code>.
     * @param blocks - the blocks that have to be deleted.
     * @throws IOException if an error occurs in the input or output stream.
     * @throws SimulatorException if an error is encountered
     * by Matlab.
     */
    private void deleteSimulinkBlocks(final String[] blocks)
        throws IOException, SimulatorException {
        for (int i = 1; i < blocks.length; i++) {
            if (blocks[i].startsWith(fileName + "/")
                    && !(blocks[i].equals(fileName + "/" + chartName))) {
                toMatlab.write("delete_block('" + blocks[i] + "')");
                toMatlab.newLine();
                toMatlab.flush();
                checkMatlabResponse();
            }
        }
    }

    /**
     * Deletes all lines in the Simulink diagram.
     * @throws IOException if an error occurs in the input or output stream.
     * @throws SimulatorException if an error is encountered
     * by Matlab.
     */
    private void deleteSimulinkLines()
        throws IOException, SimulatorException {
        toMatlab.write("sys␣=␣get_param('" + fileName + "',␣'Handle')");
        toMatlab.newLine();
        toMatlab.flush();
        checkMatlabResponse();

        toMatlab.write("delete_line(find_system(sys,'FindAll',"
                + "'on','type','line'))");
        toMatlab.newLine();
        toMatlab.flush();
        checkMatlabResponse();
    }

    /**
     * Resets the simulator.
     */
    public final void reset() {
```

```
    currentStepNumber = 0;
    createInitialPrevVariableValues();
    simBench.finishBenchmark();

    try {
        handleMatlabCommand("simin=[]");

        for (int i = 0; i < inputVariables.length; i++) {
            handleMatlabCommand(((Variable) inputVariables[i]).getName()
                    + inputVariableNamePrefix + "=[]");
        }
    } catch (Exception e) {
        log.log(0, e.getMessage());
    }
}

/**
 * Calculates the next simulation step.
 * @return A <code>MacroStep</code> object that contains the
 * simulation results.
 * @throws SimulatorException if an error is encountered
 * by Matlab.
 */
public final MacroStep nextStep() throws SimulatorException {
    if (currentStepNumber == 0) {
            simBench.reset();
    }
    simBench.start();
    String simMatrix = new String("[");
    String[] inputVarsSimMatrices = new String[inputVariables.length];
    String[] result;
    String type;
    String dataString;
    Variable var;
    Integer id;
    int nos;

    MacroStep step = new MacroStep(currentStepNumber);
    ArrayList activeStates = new ArrayList();

    //step.addMicroStep(new StateActivated(stc.getRootNode(),
    //          new Configuration(activeStates)));

    simMatrix += currentStepNumber;
    for (int j = 0; j < inputSignals.length; j++) {
        simMatrix += "␣" + signalValues[j];
    }

    simMatrix += "]";

    try {
        for (int i = 0; i < inputVariables.length; i++) {
            inputVarsSimMatrices[i] = "[" + currentStepNumber + "␣"
                + inputVarsValues[i] + "]";
            handleMatlabCommand(((Variable) inputVariables[i]).getName()
                    + inputVariableNamePrefix
                    + "=cat(1," + ((Variable) inputVariables[i]).getName()
                    + inputVariableNamePrefix
                    + "," + inputVarsSimMatrices[i] + ")");
        }
```

```
handleMatlabCommand("acs.set_param('StopTime','"
        + currentStepNumber + "')");
handleMatlabCommand("simin=cat(1,simin," + simMatrix + ")");
handleMatlabCommand("sim('" + fileName + "')");
handleMatlabCommand("mat2str(simout)");
result = input.trim().split("\\s+|;\\s*|\\[|\\]");

if (currentStepNumber == 0) {
    for (int i = 0; i < simsize; i++) {
        if (result[i + 3 + currentStepNumber * simsize]
                .matches("true|1")) {
            if (simoutArray[i] instanceof State) {
                if (!(simoutArray[i] instanceof CompositeState)) {
                    activeStates.add((State) simoutArray[i]);
                }
                step.addMicroStep(new StateActivated(
                        (State) simoutArray[i],
                        new Configuration(activeStates)));
            } else {
                step.addMicroStep(new SignalPresent(
                        (IntegerSignal) simoutArray[i]));
            }
        } else {
            if (simoutArray[i] instanceof IntegerSignal) {
                step.addMicroStep(new SignalAbsent(
                        (IntegerSignal) simoutArray[i]));
            }
        }
    }
} else {
    for (int i = 0; i < simsize; i++) {
        if (result[i + 3 + currentStepNumber * simsize]
                .matches("true|1")) {
            if ((simoutArray[i] instanceof State)
                && (!(simoutArray[i] instanceof CompositeState))) {
                activeStates.add((State) simoutArray[i]);
                if (result[i + 3 + (currentStepNumber - 1)
                            * simsize].matches("false|0")) {
                    // State was not active in previous step
                    step.addMicroStep(new StateActivated(
                        (State) simoutArray[i],
                        new Configuration(activeStates)));
                    if (simoutArray[i] instanceof IntegerSignal) {
                        step.addMicroStep(new SignalPresent(
                                (IntegerSignal) simoutArray[i]));
                    }
                } else {
                    // Event is absent if signal value
                    // has not changed.
                    if (simoutArray[i] instanceof IntegerSignal) {
                        step.addMicroStep(new SignalAbsent(
                                (IntegerSignal) simoutArray[i]));
                    }
                }
            }

        } else {
            // State is not active or Event is not present
            // in current step
```

```java
                        if ((simoutArray[i] instanceof State)
                            && (!(simoutArray[i] instanceof CompositeState))) {
                            if (result[i + 3 + (currentStepNumber - 1)
                                    * simsize].matches("true|1")) {
                                // State was active in previous step
                                step.addMicroStep(new StateDeactivated(
                                    (State) simoutArray[i]));
                                // Event is present if the signal value changes
                                if (simoutArray[i] instanceof IntegerSignal) {
                                    step.addMicroStep(new SignalPresent(
                                            (IntegerSignal) simoutArray[i]));
                                }
                            } else {
                                // Event is absent if signal value
                                // has not changed.
                                if (simoutArray[i] instanceof IntegerSignal) {
                                        step.addMicroStep(new SignalAbsent(
                                                        (IntegerSignal)
simoutArray[i]));
                                }
                            }
                        }
                    }
                }

                /*Get current values of variables*/
                for (int i = 0; i < variableNames.length; i++) {
                    result = handleMatlabCommand(variableNames[i]);
                    String value = result[2];
                    if (!value.equals((String) prevVariableValues[i])) {
                        step.addMicroStep(new VariableValue(
                                (Variable) variables[i], value));
                            prevVariableValues[i] = value;
                    }
                }
            }

            currentStepNumber++;
        } catch (IOException e) {
            throw new SimulatorException(e.getMessage());
        }
        simBench.stop();
        return step;
    }

    /**
     * Sets the statechart to the chart specified by the argument
     * <code>sc</code> and prepares the simulator for the first simulation
     * step.
     * @param sc - the new statechart that has to be simulated.
     * @throws SimulatorException if an error is encountered
     * by Matlab.
     */
    public final void setStateChart(final StateChart sc)
        throws SimulatorException {
        int[] position; // [left top right bottom ]

        stc = sc;

        try {
            fileName = (handleMatlabCommand("chart.machine.Name")[2]);
            simBench.printOutputToFile(new File("SimBench" + fileName + ".txt"));

            handleMatlabCommand("chart.Name");
            chartName = input.trim().split("\\n+")[1];
            /*check whether simulink environment is empty or not*/
            toMatlab.write("find_system('SearchDepth',1)");
            toMatlab.newLine();
            toMatlab.flush();
            checkMatlabResponse();
            String[] answer = input.trim().replaceAll("\\n+", "␣")
                .split("'\\s*'*");
            if (!(answer.length <= 4)) {
                deleteSimulinkLines();
                deleteSimulinkBlocks(answer);
            }
            enableStateOutput();
            position = getChartPosition();
            /*Set solver type to fixed-step discrete*/
            handleMatlabCommand("acs=getActiveConfigSet(model)");
            handleMatlabCommand("acs.set_param('SolverType','Fixed-step')");
            handleMatlabCommand("acs.set_param('Solver','FixedStepDiscrete')");

            handleMatlabCommand("acs.set_param('FixedStep','1.0')");

            /*Set Simulink blocks for input and output.*/
            handleMatlabCommand("simulink");

            handleMatlabCommand("add_block('simulink/Sinks/To␣Workspace','"
                    + fileName + "/output')");
            handleMatlabCommand("set_param('" + fileName
                    + "/output','SaveFormat','Array')");
            handleMatlabCommand("set_param('" + fileName
                    + "/output','Position',["
                    + (position[2] + 140) + "␣"
                    + position[1] + "␣"
                    + (position[2] + 190) + "␣"
                    + (position[1] + 25) + "])");
            noops = getNumberOfOutputPorts();
            handleMatlabCommand("add_block('simulink/Signal␣Routing/Mux','"
                    + fileName + "/route')");
            handleMatlabCommand("set_param('" + fileName
                    + "/route','Position',["
                    + (position[2] + 100 + "␣")
                    + position[1] + "␣"
                    + (position[2] + 105 + "␣")
                    + (position[1] + 55 + "])"));
            handleMatlabCommand("set_param('" + fileName
                    + "/route','Inputs','"
                    + noops + "')");

            createInputSignalsArray();
            if (inputSignals.length > 0) {
                handleMatlabCommand("add_block("
                        + "'simulink/Sources/From␣Workspace','"
                        + fileName + "/input')");
                handleMatlabCommand("set_param('" + fileName
                        + "/input','Position',["
                        + (position[2] + 200) + "␣"
                        + position[1] + "␣"
                    + (position[2] + 250) + "␣"
                    + (position[1] + 25) + "])");
            handleMatlabCommand("add_line('" + fileName
                    + "','input/1','"
                    + chartName
                    + "/Trigger')");
        }

        handleMatlabCommand("add_line('" + fileName
                + "','route/1','output/1')");

        createSimoutArray();
        //createInputSignalsArray();
        enableDataSaveToWorkspace();
        createVariableArray();
        createSimEnvForInputVariables();
        reset();
    } catch (IOException e) {
        throw new SimulatorException(e.toString());
    }
}

/**
 * Returns all input events.
 * @return A collection of all input events.
 */
public final Collection getInput() {
    return stc.getInputEvents();
}

/**
 * Returns all output events.
 * @return A collection of all output events.
 */
public final Collection getOutput() {
    return stc.getOutputEvents();
}

/**
 * Emits the event specified by the argument <code>e</code>.
 * @param e - the event.
 * @throws SimulatorException if an error is encountered
 * by Matlab.
 */
public final void emit(final Event e) throws SimulatorException {
    for (int i = 0; i < inputSignals.length; i++) {
        int port;
        if (!triggerModified) {
            try {
                setTriggerToEither();
            } catch (IOException ex) {
                throw new SimulatorException(e.toString());
            }
        }
        if ((inputSignals[i].getName()).equals(e.getName())) {
            port = inputSignals[i].getPort();
            if (signalValues[port - 1] > 0) {
                signalValues[port - 1] = -1;
            } else {
                signalValues[port - 1] = 1;
            }
        }
    }
}

/**
 * Emits the valued event specified by the argument <code>e</code>
 * with the value specified by <code>i</code>.
 * @param e - the event.
 * @param i - the value.
 * @throws SimulatorException if an error is encountered
 * by Matlab.
 */
public final void emit(final Event e, final int i)
    throws SimulatorException {
    if (triggerModified) {
        try {
            restoreTrigger();
        } catch (IOException ex) {
            throw new SimulatorException(e.toString());
        }
    }
    signalValues[((IntegerSignal) e).getPort() - 1] = i;
}

/**
 * Sets the variable specified by <code>var</code> to the value
 * specified by <code>value</code>.
 * @param var - the variable.
 * @param value - the new value of the variable.
 * @throws SimulatorException if an error is encountered
 * by Matlab.
 */
public final void setVariable(final Variable var, final String value)
    throws SimulatorException {
    for (int i = 0; i < inputVariables.length; i++) {
        if (var.equals(inputVariables[i])) {
            if (value.equals("true")) {
                inputVarsValues[i] = 1;
            } else if (value.equals("false")) {
                inputVarsValues[i] = 0;
            } else {
                inputVarsValues[i] = Double.parseDouble(value);
            }
        }
    }
}

/**
 * Sets the feature specified by <code>feature</code> to either
 * <code>true</code> or <code>false</code>. Actually, this function
 * does absolutely nothing.
 * @param feature - the feature that has to be enabled or disabled
 * @param status - specified whether to enable or disable the feature
 * @throws SimulatorException if an error is encountered
 * by Matlab.
 */
public void setFeature(final String feature, final boolean status)
        throws SimulatorException {
    }

    /**
     * Tells whether the feature <code>feature</code> is supported by this
     * simulator or not. Currently, this function returns true for the
     * 'StringLabel_Simulation' only.
     * @param feature - the feature for which one wants to know whether it is
     * supported or not.
     * @return true if the requested feature is supported, otherwise false.
     * @throws SimulatorException if an error is encountered
     * by Matlab.
     */
    public final boolean getFeature(final String feature)
            throws SimulatorException {
        if (feature.equals("StringLabel_Simulation")) {
            return true;
        } else if (feature.equals("InputEvents_Simulation")) {
            return true;
        }
            throw new SimulatorException("The␣feature␣" + feature
                                            + "is␣not␣supported");
    }
}
```

B.6.2. SimulatorException

```java
/*
 * Created on 20.10.2004
 */
package kiel.simulator;

/**
 * SimulatorException is a basic class for all exception which occur during the
 * simulation.
 * @author Andre Ohlhoff
 */
public class SimulatorException extends Exception {

   /**
    * Constructs a new SimulatorException with the specific detail message.
    * @param message
    *            the detail message.
    */
   public SimulatorException(final String message) {
      super(message);
   }
}
```

B.7. kiel.editor.controller

B.7.1. EditorModeAddJunction

```java
package kiel.editor.controller;

import kiel.dataStructure.Junction;

/**
 * The state in which the user can add a Junction.
 * <p>Copyright: (c) 2005 Adrian Posor</p>
 * <p>Company: Uni Kiel</p>
 * @author apo
 * @version $Revision: 1.1 $ last modified $Date: 2005/10/20 12:26:22 $
 */
public class EditorModeAddJunction extends EditorModeAddNode {

    /**
     * Creates an object with a Junction.gif image.
     */
    EditorModeAddJunction() {
        super("Junction.gif", Junction.class);
    }

    /**
     * @see kiel.editor.controller.EditorModeAddNode#useNodeEnlarger()
     */
    protected boolean useNodeEnlarger() {
        return true;
    }
}
```

B.7.2. ActionEditStateChartEventsForStateflow

```java
package kiel.editor.controller;

import java.awt.event.ActionEvent;
import javax.swing.JDialog;
import javax.swing.JTable;
import javax.swing.table.*;
import javax.swing.JScrollPane;
import javax.swing.JButton;
import javax.swing.JPanel;
import javax.swing.AbstractAction;
import javax.swing.JComboBox;
import javax.swing.DefaultCellEditor;
import java.util.Vector;
import java.util.ArrayList;
import java.util.Arrays;

import kiel.dataStructure.StateChart;
import kiel.dataStructure.eventexp.IntegerSignal;
import kiel.dataStructure.eventexp.Event;
import kiel.editor.resources.ResourceBundle;

/**
 * @author apo
 * @version $Revision: 1.1 $ last modified $Date: 2005/10/28 12:03:39 $
 */
public class ActionEditStateChartEventsForStateflow extends EditorAction {

    private JDialog myDialog;

    private JTable myTable;

    private DefaultTableModel myTableModel;

    private JButton okButton;

    private JButton cancelButton;

    private JButton addButton;

    private JButton deleteButton;

        /**
         * @param iconName
         */
        public ActionEditStateChartEventsForStateflow() {
                super("Edit16.gif");
        Vector columnNames = new Vector();
        columnNames.add("Name");
        columnNames.add("Scope");
        columnNames.add("Trigger");
        JComboBox triggerComboBox = new JComboBox();
        triggerComboBox.addItem("Either");
        triggerComboBox.addItem("Rising");
        triggerComboBox.addItem("Falling");
        JComboBox scopeComboBox = new JComboBox();
        scopeComboBox.addItem("Input");
        scopeComboBox.addItem("Output");
        scopeComboBox.addItem("Local");
        Vector rowData = new Vector();
        myTableModel = new DefaultTableModel(rowData, columnNames);
        myDialog = new JDialog(getEditor().getKielFrame().getJFrame(),
                ResourceBundle.getString("editStatechartEvents"), true);
        myDialog.setResizable(false);
        myDialog.setDefaultCloseOperation(JDialog.DO_NOTHING_ON_CLOSE);
        myTable = new JTable(myTableModel);
        JScrollPane myScrollPane = new JScrollPane(myTable);
        TableColumn column = myTable.getColumnModel().getColumn(2);
        column.setCellEditor(new DefaultCellEditor(triggerComboBox));
        column = myTable.getColumnModel().getColumn(1);
        column.setCellEditor(new DefaultCellEditor(scopeComboBox));
        okButton = new JButton();
        cancelButton = new JButton();
        addButton = new JButton();
        deleteButton = new JButton();
        okButton.setAction(new AbstractAction(ResourceBundle.getString("OK")) {
                public void actionPerformed(final ActionEvent e) {
                if (myTable.isEditing()) {
                    myTable.getCellEditor().stopCellEditing();
                }
                StateChart statechart = getEditor().getGraph()
                    .getMyGraphModel().getCurrentStateChart();
                IntegerSignal event;
                statechart.getInputEvents().clear();
                statechart.getOutputEvents().clear();
                statechart.getRootNode().getLocalEvents().clear();

                for (int i = 0; i < myTableModel.getRowCount(); i++) {
                    event = new IntegerSignal(
                            (String)myTableModel.getValueAt(i, 0));
                    event.setTrigger((String)myTableModel
                            .getValueAt(i, 2));
                        if (myTableModel.getValueAt(i, 1).equals("Input")) {
                        statechart.addInputEvent(event);
                    } else if (myTableModel.getValueAt(i, 1)
                            .equals("Output")) {
                        statechart.addOutputEvent(event);
                    } else {
                        statechart.getRootNode().addLocalEvent(
                                new Event((String)myTableModel
                                        .getValueAt(i, 0)));
                    }
                }
                myDialog.setVisible(false);
                getEditor().getGraph().repaint();
            }
     });
     cancelButton.setAction(new AbstractAction(ResourceBundle.getString("
Cancel")) {
            public void actionPerformed(final ActionEvent e) {
                if (myTable.isEditing()) {
                    myTable.getCellEditor().cancelCellEditing();
                }
                myDialog.setVisible(false);
            }
        });
        addButton.setAction(new AbstractAction(ResourceBundle.getString("add")) {
            public void actionPerformed(final ActionEvent e) {
                Vector newRow = new Vector();
                newRow.add("newEvent");
                newRow.add("Input");
                newRow.add("Either");
                myTableModel.addRow(newRow);
            }
        });
        deleteButton.setAction(new AbstractAction(ResourceBundle.getString("
      delete")) {
            public void actionPerformed(final ActionEvent e) {
                int[] selectedRows = myTable.getSelectedRows();
                Arrays.sort(selectedRows);
                int offset = 0;

                for (int i = 0; i < selectedRows.length; i++) {
                        myTableModel.removeRow(selectedRows[i] - offset);
                    offset++;
                }
            }
        });
        JPanel myContentPane = new JPanel();
        myContentPane.setOpaque(true);
        myContentPane.add(myScrollPane);
        myContentPane.add(addButton);
        myContentPane.add(deleteButton);
        myContentPane.add(okButton);
        myContentPane.add(cancelButton);
        myDialog.setContentPane(myContentPane);
        }


        protected void doAction(ActionEvent e) {
        getEvents();
        updateText();
        myDialog.pack();
                myDialog.setVisible(true);
        }

    private void getEvents() {
        Vector row;
        Vector dataVector = myTableModel.getDataVector();
        dataVector.clear();
        StateChart statechart = getEditor().getGraph().getMyGraphModel()
            .getCurrentStateChart();

        ArrayList events = statechart.getInputEvents();
        for (int i = 0; i < events.size(); i++) {
                row = new Vector();
            IntegerSignal event = (IntegerSignal)events.get(i);
            row.add(event.getName());
            row.add("Input");
            row.add(event.getTrigger());
            dataVector.add(row);
        }

        events = statechart.getOutputEvents();
        for (int i = 0; i < events.size(); i++) {
            row = new Vector();
            IntegerSignal event = (IntegerSignal)events.get(i);
            row.add(event.getName());
            row.add("Output");
            row.add(event.getTrigger());
            dataVector.add(row);
        }

        events = statechart.getRootNode().getLocalEvents();
        for (int i = 0; i < events.size(); i++) {
            row = new Vector();
            Event event = (Event)events.get(i);
            row.add(event.getName());
            row.add("Local");
            row.add("--");
            dataVector.add(row);
        }
        myTableModel.fireTableDataChanged();
    }

    private void updateText() {
        okButton.setText(ResourceBundle.getString("OK"));
        cancelButton.setText(ResourceBundle.getString("Cancel"));
        addButton.setText(ResourceBundle.getString("add"));
        deleteButton.setText(ResourceBundle.getString("delete"));
        myDialog.setTitle(ResourceBundle.getString("editStatechartEvents"));
    }
}
```

B.7.3. ActionEditStateChartVariablesForStateflow

```java
package kiel.editor.controller;

import java.awt.event.ActionEvent;
import javax.swing.JDialog;
import javax.swing.JOptionPane;
import javax.swing.JTable;
import javax.swing.table.*;
import javax.swing.JScrollPane;
import javax.swing.JButton;
import javax.swing.JPanel;
import javax.swing.AbstractAction;
import javax.swing.JComboBox;
import javax.swing.DefaultCellEditor;
import java.util.Vector;
import java.util.Collection;
import java.util.Iterator;
import java.util.Arrays;

import kiel.dataStructure.StateChart;
import kiel.dataStructure.boolexp.BooleanExpression;
import kiel.dataStructure.boolexp.BooleanTrue;
import kiel.dataStructure.boolexp.BooleanFalse;
import kiel.dataStructure.boolexp.BooleanVariable;
import kiel.dataStructure.doubleexp.DoubleConstant;
import kiel.dataStructure.doubleexp.DoubleVariable;
import kiel.dataStructure.floatexp.FloatConstant;
import kiel.dataStructure.floatexp.FloatVariable;
import kiel.dataStructure.int16exp.Integer16Constant;
import kiel.dataStructure.int16exp.Integer16Variable;
import kiel.dataStructure.int8exp.Integer8Constant;
import kiel.dataStructure.int8exp.Integer8Variable;
import kiel.dataStructure.intexp.IntegerConstant;
import kiel.dataStructure.intexp.IntegerVariable;
import kiel.dataStructure.uint16exp.UInt16Constant;
import kiel.dataStructure.uint16exp.UInt16Variable;
import kiel.dataStructure.uint32exp.UInt32Constant;
import kiel.dataStructure.uint32exp.UInt32Variable;
import kiel.dataStructure.uint8exp.UInt8Constant;
import kiel.dataStructure.uint8exp.UInt8Variable;
import kiel.dataStructure.Variable;
import kiel.dataStructure.Constant;
import kiel.editor.resources.ResourceBundle;
/**
 * @author apo
 * @version $Revision: 1.1 $ last modified $Date: 2005/10/28 12:03:39 $
 */
public class ActionEditStateChartVariablesForStateflow extends EditorAction {
    private JDialog myDialog;

    private JTable myTable;

    private DefaultTableModel myTableModel;

    private JButton okButton;

    private JButton cancelButton;

    private JButton addButton;

    private JButton deleteButton;

    /**
     * @param iconName
     */
    public ActionEditStateChartVariablesForStateflow() {
        super("Edit16.gif");
        Vector columnNames = new Vector();
        columnNames.add("Name");
        columnNames.add("Scope");
        columnNames.add("Data␣Type");
        columnNames.add("InitialValue");
        JComboBox typeComboBox = new JComboBox();
        typeComboBox.addItem("int32");
        typeComboBox.addItem("int16");
        typeComboBox.addItem("int8");
        typeComboBox.addItem("uint32");
        typeComboBox.addItem("uint16");
        typeComboBox.addItem("uint8");
        typeComboBox.addItem("single");
        typeComboBox.addItem("double");
        typeComboBox.addItem("boolean");
        JComboBox scopeComboBox = new JComboBox();
        scopeComboBox.addItem("Input");
        scopeComboBox.addItem("Output");
        scopeComboBox.addItem("Local");
        scopeComboBox.addItem("Constant");
        Vector rowData = new Vector();
        myTableModel = new DefaultTableModel(rowData, columnNames);
        myDialog = new JDialog(getEditor().getKielFrame().getJFrame(),
                ResourceBundle.getString("editStatechartVariables"), true);
        myDialog.setResizable(false);
        myDialog.setDefaultCloseOperation(JDialog.DO_NOTHING_ON_CLOSE);
        myTable = new JTable(myTableModel);
        JScrollPane myScrollPane = new JScrollPane(myTable);
        TableColumn column = myTable.getColumnModel().getColumn(2);
        column.setCellEditor(new DefaultCellEditor(typeComboBox));
        column = myTable.getColumnModel().getColumn(1);
        column.setCellEditor(new DefaultCellEditor(scopeComboBox));
        okButton = new JButton();
        cancelButton = new JButton();
        addButton = new JButton();
        deleteButton = new JButton();
        okButton.setAction(new AbstractAction(ResourceBundle.getString("OK")) {
            public void actionPerformed(final ActionEvent e) {
                if (myTable.isEditing()) {
                    myTable.getCellEditor().stopCellEditing();
                }
                StateChart statechart = getEditor().getGraph()
                    .getMyGraphModel().getCurrentStateChart();

                statechart.getInputVariables().clear();
                statechart.getOutputVariables().clear();
```

```
           statechart.getLocalVariables().clear();
           statechart.getConstants().clear();

           String name;
           String type;
           String initialValue;
           Variable var = null;
           Constant con = null;

           try {
                   for (int i = 0; i < myTableModel.getRowCount(); i++) {
                           name = (String)myTableModel.getValueAt(i, 0);
                           type = (String)myTableModel.getValueAt(i, 2);
                           initialValue = (String)myTableModel.getValueAt(i,
 3);

                           if (type.equals("double")) {
                                   con = new DoubleConstant(name,
                                                   Double.parseDouble(
initialValue));
                                   var = new DoubleVariable(name,
                               (DoubleConstant)con);
                           } else if (type.equals("single")) {
                                   con = new FloatConstant(name,
                                                   Float.parseFloat(
initialValue));
                                   var = new FloatVariable(name, (
FloatConstant)con);
                           } else if (type.equals("int32")) {
                                   con = new IntegerConstant(name,
                                                   Integer.parseInt(
initialValue));
                                   var = new IntegerVariable(name,
                               (IntegerConstant)con);
                           } else if (type.equals("int16")) {
                                   con = new Integer16Constant(name,
                                                   Short.parseShort(
initialValue));
                                   var = new Integer16Variable(name,
                               (Integer16Constant)con);
                           } else if (type.equals("int8")) {
                                   con = new Integer8Constant(name,
                                                   Byte.parseByte(
initialValue));
                                   var = new Integer8Variable(name,
                                                   (Integer8Constant)con);
                           } else if (type.equals("uint32")) {
                                   con = new UInt32Constant(name,
                                                   Long.parseLong(
initialValue));
                                   var = new UInt32Variable(name,
                               (UInt32Constant)con);
                           } else if (type.equals("uint16")) {
                                   con = new UInt16Constant(name,
                                                   Integer.parseInt(
initialValue));
                                   var = new UInt16Variable(name,
                               (UInt16Constant)con);
                           } else if (type.equals("uint8")) {
                                   con = new UInt8Constant(name,
                                                   Short.parseShort(
initialValue));
                                   var = new UInt8Variable(name, (
UInt8Constant)con);
                           } else if (type.equals("boolean")){
                                   if (initialValue.equals("true")) {
                                           con = new BooleanTrue(name);
                                           var = new BooleanVariable(name,
                                   (BooleanExpression)con);
                                   } else {
                                           con = new BooleanTrue(name);
                                           var = new BooleanVariable(name,
                                   (BooleanExpression)con);
                                   }
                           }
                           if (myTableModel.getValueAt(i, 1).equals("Input")
) {
                                   statechart.addInputVariable(var);
                           } else if (myTableModel.getValueAt(i, 1)
                                           .equals("Output")) {
                                   statechart.addOutputVariable(var);
                           } else if (myTableModel.getValueAt(i, 1)
                                           .equals("Local")){
                                   statechart.addVariable(var);
                           } else {
                                   statechart.addConstant(con);
                           }
                   }
           } catch (NumberFormatException nfe) {
                   JOptionPane.showMessageDialog(myDialog,
                           ResourceBundle.getString("
ErrorOnParsingTableData"),
                                           "Error", JOptionPane.
ERROR_MESSAGE);
                   return;
           }
           myDialog.setVisible(false);
           getEditor().getGraph().repaint();
       }
   });
   cancelButton.setAction(new AbstractAction(ResourceBundle
           .getString("Cancel")) {
       public void actionPerformed(final ActionEvent e) {
           if (myTable.isEditing()) {
                   myTable.getCellEditor().cancelCellEditing();
           }
           myDialog.setVisible(false);
       }
   });
   addButton.setAction(new AbstractAction(ResourceBundle
           .getString("add")) {
       public void actionPerformed(final ActionEvent e) {
           Vector newRow = new Vector();
           newRow.add("newVariable");
           newRow.add("Input");
           newRow.add("int32");
           newRow.add("0");
           myTableModel.addRow(newRow);
       }
   });
```

```
    deleteButton.setAction(new AbstractAction(ResourceBundle
            .getString("delete")) {
        public void actionPerformed(final ActionEvent e) {
            int[] selectedRows = myTable.getSelectedRows();
            Arrays.sort(selectedRows);
            int offset = 0;

            for (int i = 0; i < selectedRows.length; i++) {
                myTableModel.removeRow(selectedRows[i] - offset);
                offset++;
            }
        }
    });
    JPanel myContentPane = new JPanel();
    myContentPane.setOpaque(true);
    myContentPane.add(myScrollPane);
    myContentPane.add(addButton);
    myContentPane.add(deleteButton);
    myContentPane.add(okButton);
    myContentPane.add(cancelButton);
    myDialog.setContentPane(myContentPane);
}


protected void doAction(ActionEvent e) {
    getVariables();
    updateText();
    myDialog.pack();
    myDialog.setVisible(true);
}

private void getVariables() {
    Vector row;
    Vector dataVector = myTableModel.getDataVector();
    dataVector.clear();
    StateChart statechart = getEditor().getGraph().getMyGraphModel()
        .getCurrentStateChart();

    Collection variables = statechart.getInputVariables();
    Iterator it = variables.iterator();
    for (int i = 0; it.hasNext(); i++) {
        row = new Vector();
        Variable var = (Variable)it.next();
        row.add(var.getName());
        row.add("Input");
        row.add(getVariableType(var));
        row.add(var.toExpString());
        dataVector.add(row);
    }

    variables = statechart.getOutputVariables();
    it = variables.iterator();
    for (int i = 0; it.hasNext(); i++) {
        row = new Vector();
        Variable var = (Variable)it.next();
        row.add(var.getName());
        row.add("Output");
        row.add(getVariableType(var));
        row.add(var.toExpString());
        dataVector.add(row);
    }

    variables = statechart.getLocalVariables();
    it = variables.iterator();
    for (int i = 0; it.hasNext(); i++) {
        row = new Vector();
        Variable var = (Variable)it.next();
        row.add(var.getName());
        row.add("Local");
        row.add(getVariableType(var));
        row.add(var.toExpString());
        dataVector.add(row);
    }

    variables = statechart.getConstants();
    it = variables.iterator();
    for (int i = 0; it.hasNext(); i++) {
        row = new Vector();
        Constant con = (Constant)it.next();
        row.add(con.getName());
        row.add("Constant");
        row.add(getConstantType(con));
        row.add(con.toExpString());
        dataVector.add(row);
    }
    myTableModel.fireTableDataChanged();
}

private void updateText() {
    okButton.setText(ResourceBundle.getString("OK"));
    cancelButton.setText(ResourceBundle.getString("Cancel"));
    addButton.setText(ResourceBundle.getString("add"));
    deleteButton.setText(ResourceBundle.getString("delete"));
    myDialog.setTitle(ResourceBundle.getString("editStatechartVariables"));
}

private String getVariableType(Variable var) {
    if (var.getClass().equals(IntegerVariable.class)) {
        return "int32";
    } else if (var.getClass().equals(Integer16Variable.class)) {
        return "int16";
    } else if (var.getClass().equals(Integer8Variable.class)) {
        return "int8";
    } else if (var.getClass().equals(UInt32Variable.class)) {
        return "uint32";
    } else if (var.getClass().equals(UInt16Variable.class)) {
        return "uint16";
    } else if (var.getClass().equals(UInt8Variable.class)) {
        return "uint8";
    } else if (var.getClass().equals(FloatVariable.class)) {
        return "single";
    } else if (var.getClass().equals(DoubleVariable.class)) {
        return "double";
    } else if (var.getClass().equals(BooleanVariable.class)) {
        return "boolean";
    } else return "not␣supported␣type";
}

private String getConstantType(Constant con) {
    if (con.getClass().equals(IntegerConstant.class)) {
      return "int32";
    } else if (con.getClass().equals(Integer16Constant.class)) {
      return "int16";
    } else if (con.getClass().equals(Integer8Constant.class)) {
      return "int8";
    } else if (con.getClass().equals(UInt32Constant.class)) {
      return "uint32";
    } else if (con.getClass().equals(UInt16Constant.class)) {
      return "uint16";
    } else if (con.getClass().equals(UInt8Constant.class)) {
      return "uint8";
    } else if (con.getClass().equals(FloatConstant.class)) {
      return "single";
    } else if (con.getClass().equals(DoubleConstant.class)) {
      return "double";
    } else if (con.getClass().equals(BooleanTrue.class)
        || con.getClass().equals(BooleanFalse.class)) {
      return "boolean";
    } else return "not␣supported␣type";
  }
}
```

B.7.4. ActionEditStateEventsForStateflow

```java
package kiel.editor.controller;

import java.awt.event.ActionEvent;
import javax.swing.JDialog;
import javax.swing.JTable;
import javax.swing.table.*;
import javax.swing.JScrollPane;
import javax.swing.JButton;
import javax.swing.JPanel;
import javax.swing.AbstractAction;

import org.jgraph.graph.DefaultGraphCell;

import java.util.Vector;
import java.util.ArrayList;
import java.util.Arrays;

import kiel.dataStructure.State;
import kiel.dataStructure.eventexp.Event;
import kiel.editor.graph.MyGraphConstants;
import kiel.editor.resources.ResourceBundle;
/**
 * @author apo
 * @version $Revision: 1.1 $ last modified $Date: 2005/10/28 12:03:39 $
 */
public class ActionEditStateEventsForStateflow extends EditorAction {
    private JDialog myDialog;

    private JTable myTable;

    private DefaultTableModel myTableModel;

    private JButton okButton;

    private JButton cancelButton;

    private JButton addButton;

    private JButton deleteButton;

    /**
     * @param iconName
     */
    public ActionEditStateEventsForStateflow() {
        super("Edit16.gif");
        Vector columnNames = new Vector();
        columnNames.add("Name");
        Vector rowData = new Vector();
        myTableModel = new DefaultTableModel(rowData, columnNames);
        myDialog = new JDialog(getEditor().getKielFrame().getJFrame(),
                ResourceBundle.getString("editStateEvents"), true);
        myDialog.setResizable(false);
        myDialog.setDefaultCloseOperation(JDialog.DO_NOTHING_ON_CLOSE);
        myTable = new JTable(myTableModel);
        JScrollPane myScrollPane = new JScrollPane(myTable);
        okButton = new JButton();
        cancelButton = new JButton();
        addButton = new JButton();
        deleteButton = new JButton();
        okButton.setAction(new AbstractAction(ResourceBundle.getString("OK")) {
            public void actionPerformed(final ActionEvent e) {
                if (myTable.isEditing()) {
                    myTable.getCellEditor().stopCellEditing();
                }
                State state = (State) MyGraphConstants
                    .getGraphicalObject(((DefaultGraphCell) getEditor()
                    .getGraph().getSelectionCell()).getAttributes());
                state.getLocalEvents().clear();

                for (int i = 0; i < myTableModel.getRowCount(); i++) {
                    state.addLocalEvent(new Event(
                            (String)myTableModel.getValueAt(i, 0)));
                }
                myDialog.setVisible(false);
                getEditor().getGraph().repaint();
            }
        });
        cancelButton.setAction(new AbstractAction(ResourceBundle.getString("
Cancel")) {
            public void actionPerformed(final ActionEvent e) {
                if (myTable.isEditing()) {
                        myTable.getCellEditor().cancelCellEditing();
                }
                myDialog.setVisible(false);
            }
        });
        addButton.setAction(new AbstractAction(ResourceBundle.getString("add")) {
            public void actionPerformed(final ActionEvent e) {
                Vector newRow = new Vector();
                newRow.add("newEvent");
                myTableModel.addRow(newRow);
            }
        });
        deleteButton.setAction(new AbstractAction(ResourceBundle.getString("
delete")) {
            public void actionPerformed(final ActionEvent e) {
                int[] selectedRows = myTable.getSelectedRows();
                Arrays.sort(selectedRows);
                int offset = 0;

                for (int i = 0; i < selectedRows.length; i++) {
                    myTableModel.removeRow(selectedRows[i] - offset);
                    offset++;
                }
            }
        });
        JPanel myContentPane = new JPanel();
        myContentPane.setOpaque(true);
        myContentPane.add(myScrollPane);
        myContentPane.add(addButton);
        myContentPane.add(deleteButton);
        myContentPane.add(okButton);
        myContentPane.add(cancelButton);
        myDialog.setContentPane(myContentPane);
    }


    protected void doAction(ActionEvent e) {
        getEvents();
        updateText();
        myDialog.pack();
        myDialog.setVisible(true);
    }

    private void getEvents() {
        Vector row;
        Vector dataVector = myTableModel.getDataVector();
        dataVector.clear();
        State state = (State) MyGraphConstants
            .getGraphicalObject(((DefaultGraphCell) getEditor()
            .getGraph().getSelectionCell()).getAttributes());
        ArrayList events = state.getLocalEvents();
        for (int i = 0; i < events.size(); i++) {
            row = new Vector();
            Event event = (Event)events.get(i);
            row.add(event.getName());
            dataVector.add(row);
        }
        myTableModel.fireTableDataChanged();
    }

    private void updateText() {
        okButton.setText(ResourceBundle.getString("OK"));
        cancelButton.setText(ResourceBundle.getString("Cancel"));
        addButton.setText(ResourceBundle.getString("add"));
        deleteButton.setText(ResourceBundle.getString("delete"));
        myDialog.setTitle(ResourceBundle.getString("editStateEvents"));
    }
}
```

B.7.5. ActionEditStateVariablesForStateflow

```
package kiel.editor.controller;

import java.awt.event.ActionEvent;
import java.util.Arrays;
import java.util.Collection;
import java.util.Iterator;
import java.util.Vector;

import javax.swing.AbstractAction;
import javax.swing.DefaultCellEditor;
import javax.swing.JButton;
import javax.swing.JComboBox;
import javax.swing.JDialog;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTable;
import javax.swing.table.DefaultTableModel;
import javax.swing.table.TableColumn;

import org.jgraph.graph.DefaultGraphCell;

import kiel.dataStructure.Constant;
import kiel.dataStructure.State;
import kiel.dataStructure.Variable;
import kiel.dataStructure.boolexp.BooleanExpression;
import kiel.dataStructure.boolexp.BooleanFalse;
import kiel.dataStructure.boolexp.BooleanTrue;
import kiel.dataStructure.boolexp.BooleanVariable;
import kiel.dataStructure.doubleexp.DoubleConstant;
import kiel.dataStructure.doubleexp.DoubleVariable;
import kiel.dataStructure.floatexp.FloatConstant;
import kiel.dataStructure.floatexp.FloatVariable;
import kiel.dataStructure.int16exp.Integer16Constant;
import kiel.dataStructure.int16exp.Integer16Variable;
import kiel.dataStructure.int8exp.Integer8Constant;
import kiel.dataStructure.int8exp.Integer8Variable;
import kiel.dataStructure.intexp.IntegerConstant;
import kiel.dataStructure.intexp.IntegerVariable;
import kiel.dataStructure.uint16exp.UInt16Constant;
import kiel.dataStructure.uint16exp.UInt16Variable;
import kiel.dataStructure.uint32exp.UInt32Constant;
import kiel.dataStructure.uint32exp.UInt32Variable;
import kiel.dataStructure.uint8exp.UInt8Constant;
import kiel.dataStructure.uint8exp.UInt8Variable;
import kiel.editor.graph.MyGraphConstants;
import kiel.editor.resources.ResourceBundle;

/**
 * @author apo
 * @version $Revision: 1.1 $ last modified $Date: 2005/10/28 12:03:39 $
 */
public class ActionEditStateVariablesForStateflow extends EditorAction {
    private JDialog myDialog;

    private JTable myTable;

    private DefaultTableModel myTableModel;

    private JButton okButton;

    private JButton cancelButton;

    private JButton addButton;

    private JButton deleteButton;

    /**
     * @param iconName
     */
    public ActionEditStateVariablesForStateflow() {
        super("Edit16.gif");
        Vector columnNames = new Vector();
        columnNames.add("Name");
        columnNames.add("Scope");
        columnNames.add("Data␣Type");
        columnNames.add("InitialValue");
        JComboBox typeComboBox = new JComboBox();
        typeComboBox.addItem("int32");
        typeComboBox.addItem("int16");
        typeComboBox.addItem("int8");
        typeComboBox.addItem("uint32");
        typeComboBox.addItem("uint16");
        typeComboBox.addItem("uint8");
        typeComboBox.addItem("single");
        typeComboBox.addItem("double");
        typeComboBox.addItem("boolean");
        JComboBox scopeComboBox = new JComboBox();
        scopeComboBox.addItem("Local");
        scopeComboBox.addItem("Constant");
        Vector rowData = new Vector();
        myTableModel = new DefaultTableModel(rowData, columnNames);
        myDialog = new JDialog(getEditor().getKielFrame().getJFrame(),
                ResourceBundle.getString("editStateVariables"), true);
        myDialog.setResizable(false);
        myDialog.setDefaultCloseOperation(JDialog.DO_NOTHING_ON_CLOSE);
        myTable = new JTable(myTableModel);
        JScrollPane myScrollPane = new JScrollPane(myTable);
        TableColumn column = myTable.getColumnModel().getColumn(2);
        column.setCellEditor(new DefaultCellEditor(typeComboBox));
        column = myTable.getColumnModel().getColumn(1);
        column.setCellEditor(new DefaultCellEditor(scopeComboBox));
        okButton = new JButton();
        cancelButton = new JButton();
        addButton = new JButton();
        deleteButton = new JButton();
        okButton.setAction(new AbstractAction(ResourceBundle.getString("OK")) {
            public void actionPerformed(final ActionEvent e) {
                if (myTable.isEditing()) {
                    myTable.getCellEditor().stopCellEditing();
                }
                State state = (State) MyGraphConstants
            .getGraphicalObject(((DefaultGraphCell) getEditor()
            .getGraph().getSelectionCell()).getAttributes());
        state.getVariables().clear();
        state.getConstants().clear();

        String name;
        String type;
        String initialValue;
        Variable var = null;
        Constant con = null;

        try {
            for (int i = 0; i < myTableModel.getRowCount(); i++) {
                name = (String)myTableModel.getValueAt(i, 0);
                type = (String)myTableModel.getValueAt(i, 2);
                initialValue = (String)myTableModel.getValueAt(i, 3);

                if (type.equals("double")) {
                    con = new DoubleConstant(name,
                            Double.parseDouble(initialValue));
                    var = new DoubleVariable(name,
                            (DoubleConstant)con);
                } else if (type.equals("single")) {
                    con = new FloatConstant(name,
                            Float.parseFloat(initialValue));
                    var = new FloatVariable(name, (FloatConstant)con);
                } else if (type.equals("int32")) {
                    con = new IntegerConstant(name,
                            Integer.parseInt(initialValue));
                    var = new IntegerVariable(name,
                            (IntegerConstant)con);
                } else if (type.equals("int16")) {
                    con = new Integer16Constant(name,
                            Short.parseShort(initialValue));
                    var = new Integer16Variable(name,
                            (Integer16Constant)con);
                } else if (type.equals("int8")) {
                    con = new Integer8Constant(name,
                            Byte.parseByte(initialValue));
                    var = new Integer8Variable(name,
                            (Integer8Constant)con);
                } else if (type.equals("uint32")) {
                    con = new UInt32Constant(name,
                            Long.parseLong(initialValue));
                    var = new UInt32Variable(name,
                            (UInt32Constant)con);
                } else if (type.equals("uint16")) {
                    con = new UInt16Constant(name,
                            Integer.parseInt(initialValue));
                    var = new UInt16Variable(name,
                            (UInt16Constant)con);
                } else if (type.equals("uint8")) {
                    con = new UInt8Constant(name,
                            Short.parseShort(initialValue));
                    var = new UInt8Variable(name, (UInt8Constant)con);
                } else if (type.equals("boolean")){
                    if (initialValue.equals("true")) {
                        con = new BooleanTrue(name);
                        var = new BooleanVariable(name,
                                (BooleanExpression)con);
                    } else {
                        con = new BooleanTrue(name);
                        var = new BooleanVariable(name,
                                (BooleanExpression)con);
                    }
                }
                if (myTableModel.getValueAt(i, 1)
                        .equals("Local")){
                    state.addVariable(var);
                } else {
                    state.addConstant(con);
                }
            }
        } catch (NumberFormatException nfe) {
            JOptionPane.showMessageDialog(myDialog,
                ResourceBundle.getString("ErrorOnParsingTableData"),
                "Error", JOptionPane.ERROR_MESSAGE);
            return;
        }
        myDialog.setVisible(false);
        getEditor().getGraph().repaint();
    }
});
cancelButton.setAction(new AbstractAction(ResourceBundle
        .getString("Cancel")) {
    public void actionPerformed(final ActionEvent e) {
        if (myTable.isEditing()) {
            myTable.getCellEditor().cancelCellEditing();
        }
        myDialog.setVisible(false);
    }
});
addButton.setAction(new AbstractAction(ResourceBundle
        .getString("add")) {
    public void actionPerformed(final ActionEvent e) {
        Vector newRow = new Vector();
        newRow.add("newVariable");
        newRow.add("Local");
        newRow.add("int32");
        newRow.add("0");
        myTableModel.addRow(newRow);
    }
});
deleteButton.setAction(new AbstractAction(ResourceBundle
        .getString("delete")) {
    public void actionPerformed(final ActionEvent e) {
        int[] selectedRows = myTable.getSelectedRows();
        Arrays.sort(selectedRows);
        int offset = 0;

        for (int i = 0; i < selectedRows.length; i++) {
            myTableModel.removeRow(selectedRows[i] - offset);
            offset++;
        }
    }
});
JPanel myContentPane = new JPanel();
myContentPane.setOpaque(true);
myContentPane.add(myScrollPane);
myContentPane.add(addButton);
```

```java
        myContentPane.add(deleteButton);
        myContentPane.add(okButton);
        myContentPane.add(cancelButton);
        myDialog.setContentPane(myContentPane);
    }


    protected void doAction(ActionEvent e) {
        getVariables();
        updateText();
        myDialog.pack();
        myDialog.setVisible(true);
    }

    private void getVariables() {
        Vector row;
        Vector dataVector = myTableModel.getDataVector();
        dataVector.clear();
        State state = (State) MyGraphConstants
            .getGraphicalObject(((DefaultGraphCell) getEditor()
            .getGraph().getSelectionCell()).getAttributes());

        Collection variables = state.getVariables();
        Iterator it = variables.iterator();
        for (int i = 0; it.hasNext(); i++) {
            row = new Vector();
            Variable var = (Variable)it.next();
            row.add(var.getName());
            row.add("Local");
            row.add(getVariableType(var));
            row.add(var.toExpString());
            dataVector.add(row);
        }

        variables = state.getConstants();
        it = variables.iterator();
        for (int i = 0; it.hasNext(); i++) {
            row = new Vector();
            Constant con = (Constant)it.next();
            row.add(con.getName());
            row.add("Constant");
            row.add(getConstantType(con));
            row.add(con.toExpString());
            dataVector.add(row);
        }
        myTableModel.fireTableDataChanged();
    }

    private void updateText() {
        okButton.setText(ResourceBundle.getString("OK"));
        cancelButton.setText(ResourceBundle.getString("Cancel"));
        addButton.setText(ResourceBundle.getString("add"));
        deleteButton.setText(ResourceBundle.getString("delete"));
        myDialog.setTitle(ResourceBundle.getString("editStateVariables"));
    }

    private String getVariableType(Variable var) {
        if (var.getClass().equals(IntegerVariable.class)) {
            return "int32";
        } else if (var.getClass().equals(Integer16Variable.class)) {
            return "int16";
        } else if (var.getClass().equals(Integer8Variable.class)) {
            return "int8";
        } else if (var.getClass().equals(UInt32Variable.class)) {
            return "uint32";
        } else if (var.getClass().equals(UInt16Variable.class)) {
            return "uint16";
        } else if (var.getClass().equals(UInt8Variable.class)) {
            return "uint8";
        } else if (var.getClass().equals(FloatVariable.class)) {
            return "single";
        } else if (var.getClass().equals(DoubleVariable.class)) {
            return "double";
        } else if (var.getClass().equals(BooleanVariable.class)) {
            return "boolean";
        } else return "not␣supported␣type";
    }

    private String getConstantType(Constant con) {
        if (con.getClass().equals(IntegerConstant.class)) {
            return "int32";
        } else if (con.getClass().equals(Integer16Constant.class)) {
            return "int16";
        } else if (con.getClass().equals(Integer8Constant.class)) {
            return "int8";
        } else if (con.getClass().equals(UInt32Constant.class)) {
            return "uint32";
        } else if (con.getClass().equals(UInt16Constant.class)) {
            return "uint16";
        } else if (con.getClass().equals(UInt8Constant.class)) {
            return "uint8";
        } else if (con.getClass().equals(FloatConstant.class)) {
            return "single";
        } else if (con.getClass().equals(DoubleConstant.class)) {
            return "double";
        } else if (con.getClass().equals(BooleanTrue.class)
                || con.getClass().equals(BooleanFalse.class)) {
            return "boolean";
        } else return "not␣supported␣type";
    }
}
```

B.7.6. ActionEditStateEntryActionsForStateflow

```java
package kiel.editor.controller;
import java.awt.event.ActionEvent;

import javax.swing.JOptionPane;
import javax.swing.JTextField;

import org.jgraph.graph.DefaultGraphCell;

import kiel.dataStructure.State;
import kiel.dataStructure.action.ActionsStringLabel;
import kiel.editor.graph.MyGraphConstants;
import kiel.editor.resources.ResourceBundle;


/**
 * @author apo
 * @version $Revision: 1.1 $ last modified $Date: 2005/10/28 12:03:39 $
 */
public class ActionEditStateEntryActionsForStateflow extends EditorAction {
    /**
     * Creates an object with an Edit16.gif image.
     */
    ActionEditStateEntryActionsForStateflow() {
        super("Edit16.gif");
    }

    /**
     * @see kiel.editor.controller.EditorAction#doAction(
     * java.awt.event.ActionEvent)
     */
    protected void doAction(final ActionEvent e) {

        State state = (State) MyGraphConstants
            .getGraphicalObject(((DefaultGraphCell) getEditor().getGraph()
                .getSelectionCell()).getAttributes());
        String actionsText = state.getEntry().toString();

        final int columns = 50;
        JTextField textfield = new JTextField(actionsText, columns);
        int answer = JOptionPane.showOptionDialog(
            getEditor().getKielFrame().getJFrame(),
            textfield,
            ResourceBundle.getString("editOnEntryActions"),
            JOptionPane.OK_CANCEL_OPTION,
            JOptionPane.QUESTION_MESSAGE,
            null, new Object[] {
                ResourceBundle.getString("OK"),
                ResourceBundle.getString("Cancel")},
                ResourceBundle.getString("OK"));
        if (answer == 0) {
            state.setEntry(new ActionsStringLabel(textfield.getText()));
            getEditor().getGraph().repaint();
        }
    }
}
```

B.7.7. ActionEditStateDoActionsForStateflow

```java
package kiel.editor.controller;

import java.awt.event.ActionEvent;

import javax.swing.JOptionPane;
import javax.swing.JTextField;

import kiel.dataStructure.State;
import kiel.dataStructure.action.ActionsStringLabel;
import kiel.editor.graph.MyGraphConstants;
import kiel.editor.resources.ResourceBundle;

import org.jgraph.graph.DefaultGraphCell;


/**
 * @author apo
 * @version $Revision: 1.1 $ last modified $Date: 2005/10/28 12:03:39 $
 */
public class ActionEditStateDoActionsForStateflow extends EditorAction {
     /**
      * Creates an object with an Edit16.gif image.
      */
    ActionEditStateDoActionsForStateflow() {
        super("Edit16.gif");
    }

    /**
     * @see kiel.editor.controller.EditorAction#doAction(
     * java.awt.event.ActionEvent)
     */
    protected void doAction(final ActionEvent e) {

        State state = (State) MyGraphConstants
            .getGraphicalObject(((DefaultGraphCell) getEditor().getGraph()
                .getSelectionCell()).getAttributes());
        String actionsText = state.getDoActivity().toString();

        final int columns = 50;
        JTextField textfield = new JTextField(actionsText, columns);
        int answer = JOptionPane.showOptionDialog(
            getEditor().getKielFrame().getJFrame(),
            textfield,
            ResourceBundle.getString("EditStateDoActivities"),
            JOptionPane.OK_CANCEL_OPTION,
            JOptionPane.QUESTION_MESSAGE,
            null, new Object[] {
                ResourceBundle.getString("OK"),
                ResourceBundle.getString("Cancel")},
                ResourceBundle.getString("OK"));
        if (answer == 0) {
           state.setDoActivity(new ActionsStringLabel(textfield.getText()));
           getEditor().getGraph().repaint();
        }
    }
}
```

B.7.8. ActionEditStateExitActionsForStateflow

```java
package kiel.editor.controller;

import java.awt.event.ActionEvent;

import javax.swing.JOptionPane;
import javax.swing.JTextField;

import kiel.dataStructure.State;
import kiel.dataStructure.action.ActionsStringLabel;
import kiel.editor.graph.MyGraphConstants;
import kiel.editor.resources.ResourceBundle;

import org.jgraph.graph.DefaultGraphCell;

/**
 * @author apo
 * @version $Revision: 1.1 $ last modified $Date: 2005/10/28 12:03:39 $
 */
public class ActionEditStateExitActionsForStateflow extends EditorAction {
    /**
     * Creates an object with an Edit16.gif image.
     */
    ActionEditStateExitActionsForStateflow() {
        super("Edit16.gif");
    }

    /**
     * @see kiel.editor.controller.EditorAction#doAction(
     * java.awt.event.ActionEvent)
     */
    protected void doAction(final ActionEvent e) {

        State state = (State) MyGraphConstants
            .getGraphicalObject(((DefaultGraphCell) getEditor().getGraph()
                .getSelectionCell()).getAttributes());
        String actionsText = state.getExit().toString();

        final int columns = 50;
        JTextField textfield = new JTextField(actionsText, columns);
        int answer = JOptionPane.showOptionDialog(
            getEditor().getKielFrame().getJFrame(),
            textfield,
            ResourceBundle.getString("editOnExitActions"),
            JOptionPane.OK_CANCEL_OPTION,
            JOptionPane.QUESTION_MESSAGE,
            null, new Object[] {
                ResourceBundle.getString("OK"),
                ResourceBundle.getString("Cancel")},
                ResourceBundle.getString("OK"));
        if (answer == 0) {
           state.setExit(new ActionsStringLabel(textfield.getText()));
           getEditor().getGraph().repaint();
        }
    }
}
```

B.7.9. ActionEditStateBindActionsForStateflow

```
package kiel.editor.controller;

import java.awt.event.ActionEvent;

import javax.swing.JOptionPane;
import javax.swing.JTextField;

import kiel.dataStructure.State;
import kiel.dataStructure.action.ActionsStringLabel;
import kiel.editor.graph.MyGraphConstants;
import kiel.editor.resources.ResourceBundle;

import org.jgraph.graph.DefaultGraphCell;

/**
 * @author apo
 * @version $Revision: 1.1 $ last modified $Date: 2005/10/28 12:03:39 $
 */
public class ActionEditStateBindActionsForStateflow extends EditorAction {
    /**
     * Creates an object with an Edit16.gif image.
     */
    ActionEditStateBindActionsForStateflow() {
        super("Edit16.gif");
    }

    /**
     * @see kiel.editor.controller.EditorAction#doAction(
     * java.awt.event.ActionEvent)
     */
    protected void doAction(final ActionEvent e) {

        State state = (State) MyGraphConstants
            .getGraphicalObject(((DefaultGraphCell) getEditor().getGraph()
                .getSelectionCell()).getAttributes());
        String actionsText = state.getBindAction().toString();

        final int columns = 50;
        JTextField textfield = new JTextField(actionsText, columns);
        int answer = JOptionPane.showOptionDialog(
            getEditor().getKielFrame().getJFrame(),
            textfield,
            ResourceBundle.getString("editOnBindActions"),
            JOptionPane.OK_CANCEL_OPTION,
            JOptionPane.QUESTION_MESSAGE,
            null, new Object[] {
                ResourceBundle.getString("OK"),
                ResourceBundle.getString("Cancel")},
                ResourceBundle.getString("OK"));
        if (answer == 0) {
           state.setBindAction(new ActionsStringLabel(textfield.getText()));
           getEditor().getGraph().repaint();
        }
    }
}
```

B.8. kiel.editor.graph

B.8.1. JunctionRenderer

```java
package kiel.editor.graph;

import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.Graphics2D;

import kiel.editor.MyJGraph;

/**
 * Describes a Junction in the JGraph model.
 * <p>Copyright: (c) 2005 Adrian Posor</p>
 * <p>Company: Uni Kiel</p>
 * @author apo
 * @version $Revision: 1.1 $ last modified $Date: 2005/10/20 12:26:22 $
 */
public class JunctionRenderer extends PseudoStateRenderer {

    protected void paintSelectionBorder(final Graphics g) {
        ((Graphics2D) g).setStroke(getGraph().getStroke(getView()));
        Dimension d = getSize(); g.setColor(
            ((MyJGraph) graph).getViewsBlack(view));
        g.drawOval(0, 0, d.width - 1, d.height - 1);
    }
}
```