Christian-Albrechts-Universität zu Kiel

Student Research Project

Simulating the Behavior of SyncCharts

André Ohlhoff

February 5, 2006

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

This student research project deals with the development of a simulator for the behavior of SyncCharts. SyncCharts are a Statechart variant based on the synchronous language Esterel. The commercial variant of SyncCharts is called Safe State Machines and used in *Esterel Studio*. The simulator is a part of the *KIEL* project and controlled by the *browser* via a very small interface. The results of the simulation do not only contain enough information to allow developers to simulate their syncCharts but also to help novices to understand the semantics of SyncCharts in the first place.

Contents

1 Introduction 7

2 SyncCharts 9

2.1 Signals and Variables . . . 10

2.1.1 Signals . . . 10

2.1.2 Variables . . . 11

2.2 States . . . 11

2.2.1 Normal States . . . 11

2.2.2 Pseudo States . . . 12

2.3 Transitions . . . 14

2.3.1 Transition Labels . . . 14

2.3.2 Transition Types . . . 17

2.4 State Actions . . . 17

3 Simulator 19

3.1 The KIEL Project . . . 19

3.1.1 History Connector . . . 19

3.2 Constructiveness . . . 20

3.2.1 Reactivity . . . 20

3.2.2 Determinism . . . 21

3.2.3 Logical Correctness . . . 21

3.2.4 Constructiveness . . . 21

3.3 The Simulator . . . 23

3.3.1 The Structure . . . 24

3.3.2 The Results . . . 24

3.3.3 The Reaction . . . 25

3.3.4 The Potential . . . 25

4 Implementation 27

4.1 The Package kiel.simulator . . . 27

4.1.1 The Class SimulatorException . . . 27

4.1.2 The Interface Simulator . . . 27

4.2 The Package kiel.configMngr . . . 28

4.2.1 MicroSteps and MacroSteps . . . 29

4.3 The Package kiel.simulator.syncchart.converter . . . 30

4.4 The Package kiel.simulator.syncchart.exceptions . . . 30

4.5 The Package kiel.simulator.syncchart.intexp 31
4.5.1 The Interface STRLIntegerExpression 31
4.5.2 The Class STRLVariableHandler 32
4.6 The Package kiel.simulator.syncchart.boolexp 32
4.6.1 The Class STRLPre 32
4.7 The Package kiel.simulator.syncchart.sigexp 34
4.7.1 The Class STRLSignal 34
4.7.2 The Class STRLPureSignal 35
4.7.3 The Class STRLSignalTick 35
4.7.4 The Class STRLSignalTrue 35
4.7.5 The Class STRLIntegerSignal 35
4.7.6 The Class STRLSignalHandler 36
4.8 The Package kiel.simulator.syncchart.action 36
4.8.1 The Class STRLAction 36
4.8.2 Delayed Action Handling 38
4.9 The Package kiel.simulator.syncchart.transition 39
4.9.1 The Class STRLDelayExpression 41
4.9.2 The Class STRLTransition 41
4.9.3 The Class STRLTransitionHandler 42
4.10 The Package kiel.simulator.syncchart 42
4.10.1 The Class STRLReactingComponent 44
4.10.2 The Class STRLSimulator 44
4.10.3 The Class STRLState 46
4.10.4 The Class STRLSimpleState 46
4.10.5 The Class STRLMacroState 46
4.10.6 The Class STRLSTG 48
4.10.7 The Class STRLSTGHandler 49
4.10.8 The Class STRLSuspend 49
4.10.9 The Class STRLReactiveCell 49
4.10.10 The Class STRLPseudoCell 49
4.10.11 The Class STRLStateCell 50

5 Future Work 55

6 Conclusions 57

Bibliography 59

Source Code 61
.1 kiel.configMngr 61
.1.1 `ChangeValue.java` 61
.1.2 `ConfigMngr.java` 63
.1.3 `Configuration.java` 65
.1.4 `ExecuteTransition.java` 67

.1.5 MacroStep.java 68
.1.6 MicroStep.java 69
.1.7 OnEntry.java 70
.1.8 OnExit.java 71
.1.9 OnInside.java 72
.1.10 SignalAbsent.java 73
.1.11 SignalPresent.java 74
.1.12 SignalStatus.java 75
.1.13 SignalUnknown.java 76
.1.14 SignalValue.java 77
.1.15 StateActivated.java 78
.1.16 StateDeactivated.java 79
.1.17 StateNotSuspended.java 80
.1.18 StateStatus.java 81
.1.19 StateStatusAndConfig.java 82
.1.20 StateSuspended.java 83
.1.21 StateTransient.java 84
.1.22 TestTransition.java 85
.1.23 TransitionStatus.java 86
.1.24 VariableValue.java 87
.2 kiel.simulator 88
.2.1 Simulator.java 88
.2.2 SimulatorChoser.java 89
.2.3 SimulatorException.java 90
.3 kiel.simulator.syncchart 91
.3.1 STRLComponent.java 91
.3.2 STRLConditionalCell.java 92
.3.3 STRLInitialCell.java 93
.3.4 STRLMacroState.java 94
.3.5 STRLPseudoCell.java 97
.3.6 STRLReactingComponent.java 100
.3.7 STRLReactiveCell.java 102
.3.8 STRLSimpleState.java 103
.3.9 STRLSimulator.java 105
.3.10 STRLState.java 110
.3.11 STRLStateCell.java 112
.3.12 STRLSTG.java 118
.3.13 STRLSTGHandler.java 121
.3.14 STRLSuspend.java 123
.3.15 SyncChart.java 124
.4 kiel.simulator.syncchart.action 126
.4.1 STRLAction.java 126
.4.2 STRLActions.java 127
.4.3 STRLAssignment.java 128

.4.4 STRLDelayedAction.java 129
.4.5 STRLDelayedActionHandler.java 131
.4.6 STRLEmitSignal.java 132
.4.7 STRLEmitValuedSignal.java 133
.4.8 STRLInitializeSignal.java 134
.4.9 STRLInitializeTransition.java 135
.4.10 STRLInitializeVariable.java 136
.5 kiel.simulator.syncchart.boolexp 137
.5.1 STRLAnd.java 137
.5.2 STRLBooleanExpression.java 139
.5.3 STRLBrackets.java 140
.5.4 STRLComparator.java 141
.5.5 STRLEqual.java 142
.5.6 STRLFalse.java 143
.5.7 STRLGreaterOrEqual.java 144
.5.8 STRLGreaterThan.java 145
.5.9 STRLLessOrEqual.java 146
.5.10 STRLLessThan.java 147
.5.11 STRLNot.java 148
.5.12 STRLNotEqual.java 149
.5.13 STRLOr.java 150
.5.14 STRLTrue.java 152
.6 kiel.simulator.syncchart.converter 153
.6.1 ActionConverter.java 153
.6.2 ActionsConverter.java 154
.6.3 ANDToMacro.java 155
.6.4 CompConverter.java 156
.6.5 Context.java 157
.6.6 DelayExpressionConverter.java 161
.6.7 DynChoiceToCondCell.java 162
.6.8 GuardConverter.java 163
.6.9 ImmediateTransConv.java 164
.6.10 InitStateToInitCell.java 165
.6.11 IntExpConverter.java 166
.6.12 KielToSyncChart.java 167
.6.13 NormTermConv.java 168
.6.14 ORToMacro.java 169
.6.15 RegionConverter.java 170
.6.16 SimpleToSimple.java 173
.6.17 StateToStateCell.java 174
.6.18 StrongAbortConv.java 175
.6.19 TransConverter.java 176
.6.20 TriggerConverter.java 177
.6.21 WeakAbortConv.java 178

.7 kiel.simulator.syncchart.exceptions 179
.7.1 STRLException.java 179
.7.2 STRLIntegerSignalNotFoundException.java 180
.7.3 STRLNotSupportedException.java 181
.7.4 STRLSignalNotFoundException.java 182
.7.5 STRLUninitializedReadException.java 183
.7.6 STRLVariableNotFoundException.java 184
.8 kiel.simulator.syncchart.intexp 185
.8.1 STRLIntegerAdd.java 185
.8.2 STRLIntegerBrackets.java 186
.8.3 STRLIntegerConstant.java 187
.8.4 STRLIntegerDiv.java 188
.8.5 STRLIntegerExpression.java 189
.8.6 STRLIntegerMinus.java 190
.8.7 STRLIntegerMod.java 191
.8.8 STRLIntegerMult.java 192
.8.9 STRLIntegerOperator.java 193
.8.10 STRLIntegerSub.java 194
.8.11 STRLIntegerVariable.java 195
.8.12 STRLVariableHandler.java 197
.9 kiel.simulator.syncchart.sigexp 198
.9.1 STRLCombineSignal.java 198
.9.2 STRLCombineWithAdd.java 200
.9.3 STRLCombineWithMult.java 201
.9.4 STRLIntegerSignal.java 202
.9.5 STRLPre.java 204
.9.6 STRLPureSignal.java 206
.9.7 STRLSignal.java 207
.9.8 STRLSignalHandler.java 209
.9.9 STRLSignalTick.java 211
.9.10 STRLSignalTrue.java 212
.9.11 STRLSingleSignal.java 213
.10 kiel.simulator.syncchart.transition 214
.10.1 STRLDelayExpression.java 214
.10.2 STRLImmediateTrans.java 217
.10.3 STRLNormalTermination.java 218
.10.4 STRLStrongAbortion.java 219
.10.5 STRLTransition.java 220
.10.6 STRLTransitionHandler.java 223
.10.7 STRLWeakAbortion.java 225

1 Introduction

The development of reactive and embedded systems, such as an airbag controller, is a hard task. Many target devices only generate limited feedback. Hence, it is very difficult to locate bugs, if the device does not work as intended. The introduction of state based models, such as Harel's Statecharts [10, 9], greatly improved the process of developing reactive systems. Very important and valuable is the possibility to simulate the system even before transferring the program to the target. However, it stays difficult to keep track of always increasing models. Today the popular tools use only a static view during the editing and simulation process. It is up to the user to follow the control flow.

The main goal of the *KIEL* (**K**iel **I**ntegrated **E**nvironment for **L**ayout) project [14] is to improve the aspects of model design with automatic layout and dynamic elements. Using dynamic views during the simulation process makes the behavior of the models much more comprehensible. This student research project has the particular task to add a simulator module to the KIEL project that simulates the behavior of SyncCharts. The commercial variant of SyncCharts is called *Safe State Machines* (SSM) and used in *Esterel Studio* [8]. The KIEL *browser* [15] controls the simulator via a straightforward interface and receives a lot of information during the simulation process. The simulator assumes that every syncChart is proved to be syntactical correct before the simulation starts. Even though the simulator should only be applied to *constructive* syncChart, it needs dynamic constructive checking anyway. Section 3.2 gives a short introduction to constructiveness.

2 SyncCharts

At first glance a syncChart looks like a statechart described by Harel [10, 9]. However, the resulting behavior of Statecharts and SyncCharts is different.

SyncCharts (synchronous charts) were introduced in 1996 by *Charles André* [2, 1] as a graphical notation for the synchronous language Esterel [7, 6]. A syncChart describes the behavior of a reactive system. Following the synchronous hypothesis of Esterel the reaction takes place in successive discrete instants. The system reacts only at these instants and is idle in between. Each instant is divided in the three phases *read input*, *compute reaction* and *generate output*. It is assumed that the reaction is computed in zero time, so that the output is generated simultaneous with the input. Actually the output might have an impact on the computation itself.

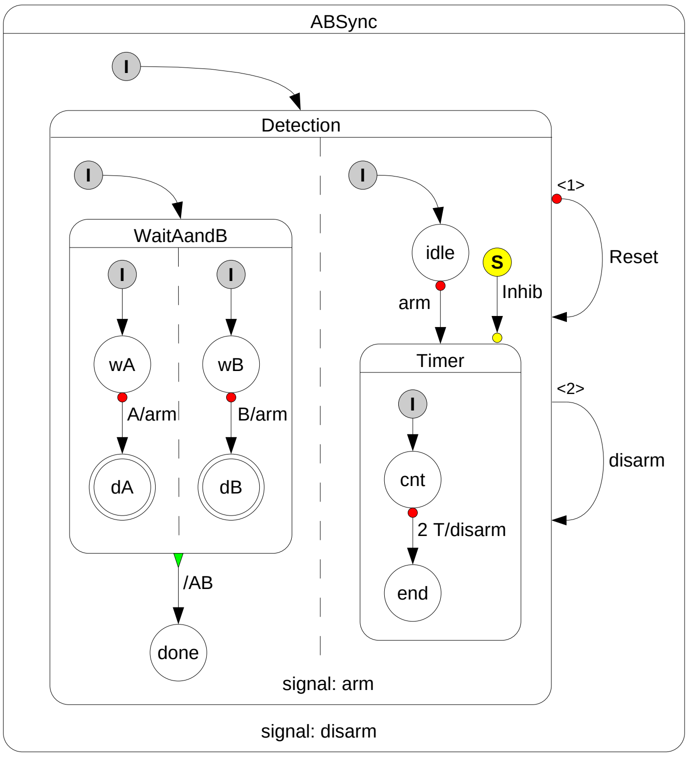


Figure 2.1: A typical syncChart

Figure 2.1 is taken from *Computing SyncCharts Reactions*[3] and shows a typical syncChart. If the reader is not familiar with the concept of state based models it might be better to skip the following description of the behavior until the end of this chapter. The syncChart *ABSync* has five input signals (*A*, *B*, *T*, *Reset* and *Inhib*) and only one output signal (*AB*). Nothing special happens during the first instant. The states *wA*, *wB* and *idle* become active and no other transition can be enabled because they are not immediate. If the signal *Reset* becomes present in the following instants, the syncChart will be reset to this initial situation ignoring the status of the other signals. After both *A* and *B* were present at least once, the state *WaitAandB* will take its normaltermination transition and emit the output signal *AB*. The first occurrence of *A* or *B* also activates the state *Timer* that waits with its simple state *cnt* from the following instant for occurrences of *T*. If *T* was present twice the signal *disarm* is emitted, which resets the syncChart again to its initial situation. Because of the weakabortion transition the reset via *disarm* allows the emission of *AB*, if the requirements are fulfilled. The suspension with the signal *Inhib* will disable the counting of *T* every instant that *Inhib* is present.

2.1 Signals and Variables

Signals are one of the most important concepts of SyncCharts. Emitting and receiving signals is used to model the communication and synchronization between the system and its environment and also between concurrent threads of the same system. Each signal has a scope. *Interface signals* are visible within the whole model and can further be separated in *input* and *output* signals. *Local signals* are only visible within the surrounding *macro state*. In figure 2.1 *A*, *B*, *T*, *Reset* and *Inhib* are input signals, *AB* is the only output signal and *arm* and *disarm* are local signals defined in *Detection* and *ABSync* respectively.

2.1.1 Signals

Every signal has a *presence status*. A signal is either *present* or *absent*. During the computation a third status *unknown* is used until the real status is revealed. SyncCharts use the instantaneous broadcast mechanism. If the status of a signal changes, every component within the scope of this signal is informed immediately. Every instant leads to a well-defined presence status (i.e. either *present* or *absent*) for each signal. Otherwise the model is not considered to be *constructive* (see 3.2).

Pure Signals

Pure signals are the basic signals. They only carry their presence status. There is a predefined pure signal *tick* that is present every instant.

Valued Signals

Valued signals have both a presence status and an integer value and can further be separated in *single signals* and *combine signals*. A single signal can only be emitted with a valid value and only once during one instant. Several simultaneous emissions of the same single signal would not lead to a well defined value and are considered a run-time error. *Combine signals* use an associative and commutative *combination function* to allow multiple emissions within the same instant. The simulator supports multiplication and addition as combination functions. For example, an additive combine signal is emitted with the values *5* and *6*, the current value of this signal is *11*.

2.1.2 Variables

A syncChart can also contain integer *variables*. Unlike valued signals a variable can not be shared between different threads, but the value of a variable can change arbitrary times during an instant. Variables are mostly used to save intermediate results. The scope of a variable is the surrounding *region* (see macro states 2.2.1). The simulator does not check for concurrent read/write issues. For example, figure 2.2 contains a variable.

2.2 States

At first SyncCharts distinguishes between *normal states* and *pseudo states*. As opposed to normal states the simulation will never rest in a pseudo state.

2.2.1 Normal States

Normal states are either *simple states* or *macro states*.

Simple States

Simple states are usually drawn as a circle or an ellipse. A simple state may contain a label with its name. For example, in figure 2.1 *wA* is a simple state.

Macro States

A *macro state* is drawn as a rounded rectangle with a small header. The header may contain a label with the macro state's name. A simple state is not refined, but a macro state must contain at least one *region*. A region is a connected set of states and exact one *initial state*. Several regions within the same macro state are separated with dashed lines. For example, in figure 2.1 *Detection* is a macro state with two regions.

Final States

A simple state/macro state can be tagged as *final*. A *final simple state*/*final macro state* is drawn with a double outline. In figure 2.1 *dA* and *dB* are final simple states.

Final states do not have outgoing transitions.

2.2.2 Pseudo States

In addition to normal states, there are several types of *pseudo states*.

Initial States

An *initial state* is drawn as a small usually gray circle labeled with an *I*. An initial state is used as the entry point for a region. Figure 2.1 contains six initial states. Initial states do not have incoming transitions.

Conditional States

A *conditional state* is drawn similar to a initial state but is labeled with a *C*. Conditional states are used to merge and link different transitions in order to remove redundancy. The syncChart *Gum_ Vending_ Machine* in figure 2.2 contains a conditional state. It describes a very simple gum vending machine. A gum costs 15 cent and only 5 and 10 cent coins are permitted. Each inserted coin triggers a new instant with either *FIVE* or *TEN* as input. The system resides in the state *wait* and the signal *FIVE* will increase the value of *v* by 5, the signal *TEN* will increase the value of *v* by 10. If *v* is now greater or equal to 15 the transition with the emission of *PAID* is taken. Otherwise the default transition will be enabled. At the end of each instant the state *wait* is active and waits for the next input. The presence of *PAID* leads to a weakabortion and a reset of *Compute* and the emission of the output signal *GUM*. There is no a possibility to get the change back.

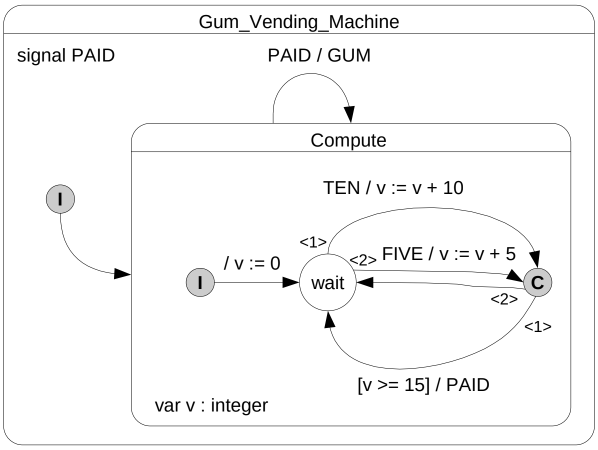


Figure 2.2: Controller for a simple gum vending machine.

Suspend Connectors

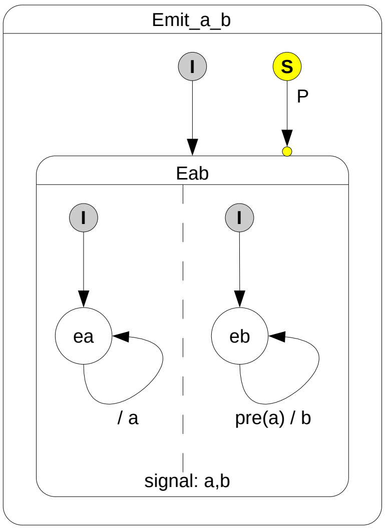


Figure 2.3: The pre operator is not affected by the suspend connector.

The yellow circle labeled with an S in figure 2.3 is a *suspend* connector. The arrow from the connector points to the state that can be suspended. The arrow can be labeled with the optional immediate modifier (2.3.1) and a signal expression (2.3.1). If the trigger is satisfied the body of the target state is suspended. Outgoing transition can still be taken. The syncChart *Emit_ a_ b* in figure 2.3 does not emit anything in its first instant. In the second instant the signal a is emitted and after that a and b are emitted every following instant. If the signal P is present the state *Eab* is suspended and its body does not react. If the state is resumed in a following instant, the trigger *pre(a)* does not relate to the absolute previous instant, but to the previous instant, where the state (and the signal a) was not suspended.

History Connectors

Usually a *history* connector is drawn as a small gray circle labeled with *H**. It can be attached to the outline of a macro state. Incoming transitions pointing at the history connector instead of the macro state itself, will resume the execution of the macro state right at time it was exited. If the macro state is reached for the first time the history connector has no effect. For example, figure 2.4 contains a history connector. Macro states with a history connector are treated like suspended states. If a state with history is reentered and the pre operator is applied to its local signals, then it refers to the status of the signal in the last instant where the state was active.

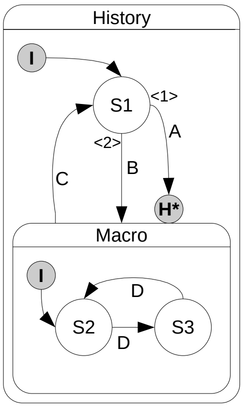


Figure 2.4: The history connector

2.3 Transitions

Transitions advance the system. An enabled transition might change the current active state. Transitions are drawn as arrows, pointing from a source state to a target state. Self loops are allowed. Each state (normal and pseudo) can have an arbitrary number of outgoing transitions. If there is more than one transition all transitions must be labeled with a priority number (1 is the highest priority). If several transitions can be enabled during one instant, the one with the highest priority is taken. SyncCharts do not allow inter-level transitions.

2.3.1 Transition Labels

The general form of the transition label is
`# <countdelay> <signalexpression> [<guard>] / <listofactions>`.
Each part is optional. The *immediate modifier* `#` and the countdelay are exclusive.

Signal Expressions

A *signal expression* is a Boolean expression over signal statuses. If a signal expression is satisfied the corresponding transition can be enabled. The signal expression `A and not B` is satisfied, if *A* is present and *B* is absent. An empty signal expression is always satisfied.

Aside from the Boolean operators *and*, *or* and *not* the signal expression may also contain the *pre* operator. The signal expression `A and not pre(B)` is satisfied, if A is present and B was not present in the previous instant.

Immediate modifier

Transitions in SyncCharts are not enabled in the same instant the source state was reached. However, the immediate modifier `#` is used to change this behavior. If a strongabortion transition is activated immediately in the first instant of a state, this state is *transient*. An immediate loop (containing one or more states and transitions) is not allowed. The syncChart *Reincarnation* in figure 2.5 uses the immediate modifier. This example looks very simple, but the behavior is quite surprising. *A*, *B*, *C* and *D* are pure input signals, *V* is a multiplicative combine signal. In the first instant the state *S1* becomes active and *V* is emitted with the value 2. If all four input signals are present in the next instant *S3* will become the active state and the value of *V* is 11550. Each transition emits *V* with a different prime number so the result can be decomposed to `2*3*5*5*7*11`. The following has happened:

1. *A* triggers the self loop of *S1*. *V* is emitted with 3.
2. The same transition cannot be enabled again, but the second transition is immediate and triggered by *B* so that *S1* becomes transient. *V* is emitted with 5.
3. The weak self loop of *InnerMacro* can be taken, because the body has completed its reaction. *V* is emitted with 7.
4. The initial state passes control to *S1* again. *V* is emitted with 2.
5. The immediate transition between *S1* and *S2* is taken. *S1* becomes transient again and *V* is emitted with 5.
6. Finally, the only transition that can be enabled is the weakabortion towards *S3*. *V* is emitted with 11.

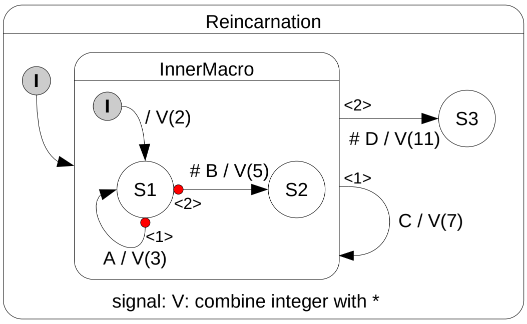


Figure 2.5: A syncChart containing a multiplicative combine signal, self loops and immediate modifier.

Count Delays

A *count delay* is an integer expression that specifies how many instants the signal expression must be satisfied before the transition can be enabled. The syncChart *ABSync* in figure 2.1 uses a count delay to wait for 2 occurrences of *T*.

Guards

A *guard* is a Boolean expression that must be true in the moment the transition is triggered. The simulator supports Boolean expression such as
`((X > 0) and (?A + ?B) = 10)`

Actions

Each transition can execute a list of actions. The simulator supports basic integer arithmetic using `+`, `-`, `*`, `div`, `mod` and the `pre` operator for valued signals. See 2.4 for a detailed description of actions associated with states.

`A, B:` :
    Emits the pure signals A and B.

`X := Y + ?C:` :
    Assigns the sum of the value of the variable Y and the valued signal C to the variable X.

`D( 5 ):` :
    Emits the valued signal D with the value `5`.

`E( pre(?E) * 2 ):` :
    Emits the valued signal E with the double value of the previous instant.

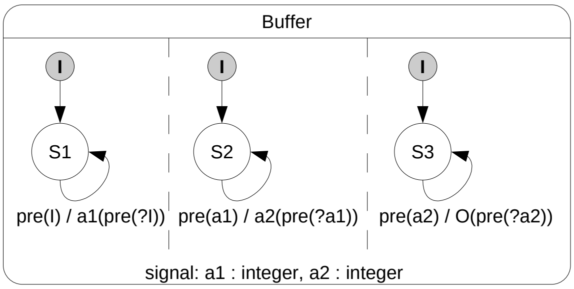


Figure 2.6: A simple integer buffer using the pre operator

Figure 2.6 shows a syncChart using the pre operator to describe a simple integer buffer. The buffer has one valued input signal I and one valued output signal O. The left-most region will emit the valued local signal *a1* if the signal I was present in the previous instant. The previous value of I will become the current value of *a1*. The other regions behave in the same way but use different signals. So every emission of I will lead to an emission of O exactly three instants later with the same value.

2.3.2 Transition Types

There are three main types of transitions that differ from each other in the way they are activated.

Strongabortion

Strongabortion transitions have the highest priority and are evaluated before the evaluation of the source state and its body. However, strongabortion transitions without an immediate modifier are not evaluated in the first instant of the source state. Strongabortion transitions are tagged with a small red circle at the source state. In figure 2.1 the transition with the label `Reset` is a strongabortion transition.

Weakabortion

Weakabortion transitions are drawn as normal arrows. They are evaluated after the evaluation of the source and its body. In figure 2.1 the transition with the label `disarm` is a weakabortion transition.

Normaltermination

Normaltermination transitions have the lowest priority. Normaltermination transitions cannot be labeled with a signalexpression, countdelay, immediate modifier or guard and can only have macro states as source states. A normaltermination transition is enabled, if all regions of the source state have reached a final state. If the source state terminates right after he is reached, the transition is taken immediately (as long as it is not overridden by another immediate transition). Normaltermination transitions are drawn with a green triangle at the source state. For example, in figure 2.1 the transition with the label `/AB` is a normaltermination transition that is enabled if both *dA* and *dB* are reached.

Immediate Transitions

Transitions starting from a pseudo state are implicit immediate and cannot have a countdelay. As every state, pseudo states are allowed to have an arbitrary number of outgoing transitions. However, there must be a *default transition*[1], so that the system can leave the pseudo state in any case.

2.4 State Actions

In addition to actions that are executed when a transition is enabled there are three types of actions that are optional directly linked to a *normal state*. Before these actions can be explained, some terms need to be defined first.

[1] A default transition is a transition with the lowest priority and neither trigger nor guard

- A state is *entered* (also called *activated*), if the state is the target of an enabled transition and if the state does not turn out to be *transient*.
- A state if *active* during an instant, if the state
  - was entered in the same instant or if the state
  - was active at the end of the previous instant and does not enabled a strongabortion transition during this instant.
- A state is *exited* (also called *deactivated*), if the state was active and enables a transition. If a state without a history connector is exited, his active child states are exited before the parent state.

The three state actions differ in the way they are executed:

OnEntry Actions :
A state executes its onentry actions, when a state is entered.

OnInside Actions :
A state executes its oninside actions, when the state is active.

OnExit Actions :
A state executes its onexit actions, when the state is exited.

Due to the weakabortion transitions, a state might become active twice during one instant and also executes its oninside actions twice. Final states do not have oninside actions or onexit actions.

3 Simulator

The previous chapter describes the elements of SyncCharts as far as they are supported by the simulator. This chapter gives a raw overview of how these elements are represented. In addition section 3.1 contains a short description of the KIEL project. Section 3.2 introduces the concept of *constructiveness*. Section 3.3.4 describes how the computation of a *potential* (i.e. the set of signals that still can be emitted) can resume a frozen simulation. Finally, chapter 4 contains detailed information about the implementation.

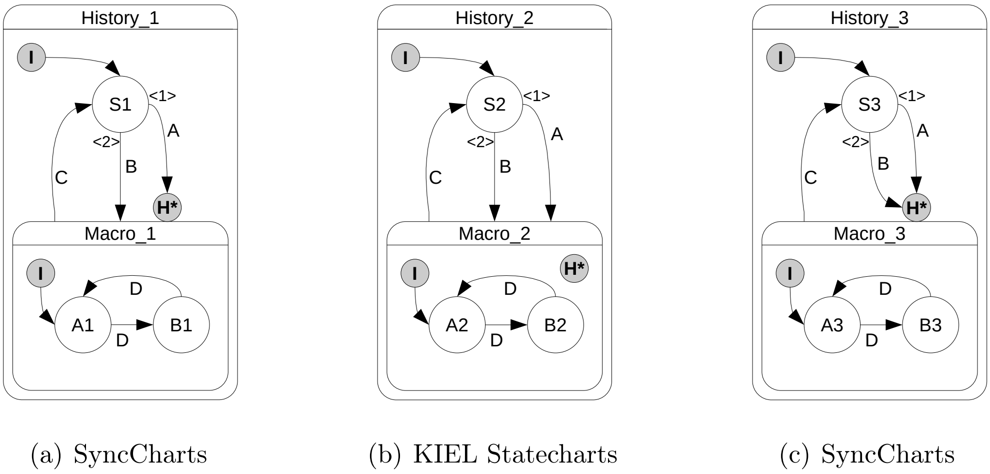
3.1 The KIEL Project

KIEL (**K**iel **I**ntegrated **E**nvironment for **L**ayout) is the name for a new statechart modeling tool. Its main goal is to enrich the development of statecharts with dynamic elements. The design of KIEL is very modular. At the moment, KIEL consists of a *layouter* [11], an *editor* [13], a *browser* [15] and several *fileinterfaces* [16, 12] to import statecharts from other tools.

The core of the KIEL project is a data structure based on UML statecharts. This structure holds only the syntactical parts of a syncChart. The design is straightforward. There are *nodes* and *edges* that are refined to *states*, *pseudostates*, *transitions* and many different types of labels. Everything mentioned in the previous chapter can be represented. Graphical information is recorded in additional classes, but are not necessary for the simulation.

3.1.1 History Connector

Due to the *UML* paradigm, it is not possible to realize the history connector exactly. Since two states can only be connected by a transition and the layouter can not handle states directly attached to each other, the history connector becomes a child of the relevant macro state.



(a) SyncCharts (b) KIEL Statecharts (c) SyncCharts

Figure 3.1: Differences between SyncCharts and KIEL Statecharts.

Figure 3.1(b) shows how this looks like in KIEL. The simulator will treat this history connector as if every incoming transition points at it (see 3.1(c)). It is not possible to have several incoming transitions that use the history connector and some that do not. The syncChart 3.1(a) can not be realized in KIEL.

3.2 Constructiveness

Before the simulator is explained. The concept of *constructiveness* needs to be mentioned. Unlike common programming languages the synchronous approach of SyncCharts with the instantaneous broadcast of signals implies the possibility to create syntactically correct syncCharts that must be rejected because of semantic problems. Two basic requirements are *reactivity* and *determinism*.

3.2.1 Reactivity

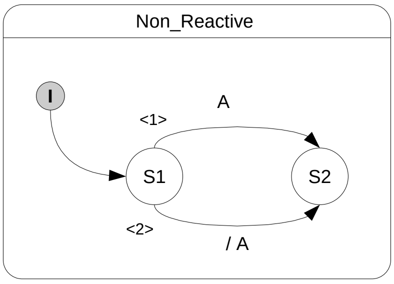


Figure 3.2: A non-reactive syncChart

A syncChart is *reactive*, if it produces output for each input. Figure 3.2 shows a very simple non-reactive syncChart. It can not be decided whether to take the upper or lower transition. The upper transition can only be enabled if A is present. The lower transition has no trigger, but if it is enabled A becomes present and so the upper transition should rather be enabled because of the higher priority.

3.2.2 Determinism

A syncChart is *deterministic*, if it produces only one output for each input. Figure 3.3 shows a non-deterministic syncChart. Both the execution of the upper and lower transition seem to lead to a well-defined signal status with A present or absent respectively.

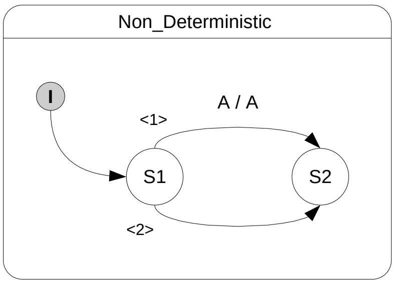


Figure 3.3: A non-deterministic syncChart

3.2.3 Logical Correctness

A syncChart that is both reactive and deterministic is *logical correct*. Common to all erroneous syncCharts is that they contain some kind of cyclic signal dependency that can not be resolved. The syncChart `Non_Reactive` requires to solve `A = not A`. The problem in `Non_Deterministic` is equivalent to `A = A`, which allows more than one unique solution. The dependency can also occur between an arbitrary number of signals and valued signals. The syncChart in figure 3.4 is again not deterministic because `((A = B) and (B = A))` does not have a unique solution.

The intention of the syncChart in figure 3.5 is to increase the value of O every instant. But the instantaneous broadcast mechanism leads to the not-solvable problem `?O = ?O + 1`. The syncChart would work as intended with the pre operator. Figure 3.6 shows a similar problem involving two valued signals.

3.2.4 Constructiveness

Figure 3.7 shows why SyncCharts demands even more than logical correctness. This syncChart is logical correct. The absence of both A and B in the second and every

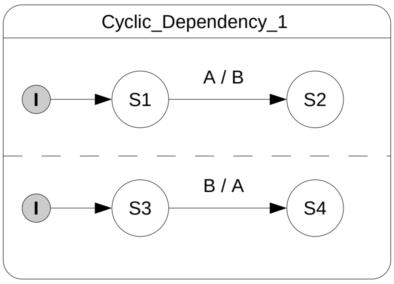


Figure 3.4: Cyclic dependency of A and B

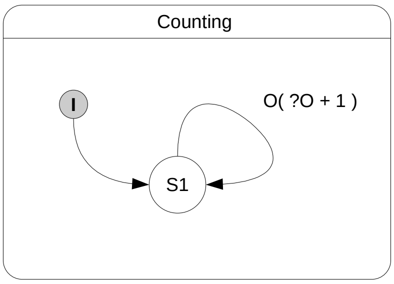


Figure 3.5: Self dependency of a valued signal

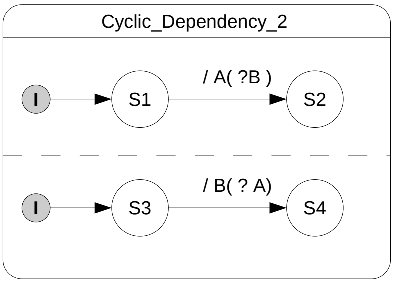


Figure 3.6: Cyclic dependency of valued signals

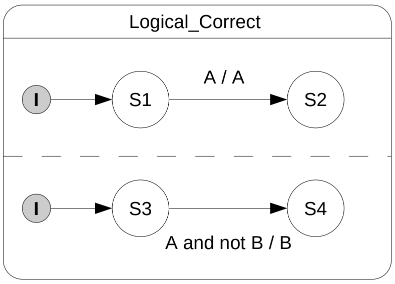


Figure 3.7: A surprisingly logical correct syncChart

following instant conforms to the logical correctness. But this is not satisfying at all and that is why the constructive semantics of SyncCharts forbids any kind of self-justification or speculative computing. *The Esterel v5 Language Primer*[5] establishes several rules for pure Esterel. These rules can be adapted to pure SyncCharts as follows:

1. All signals are initially set to unknown.
2. Input signals that are emitted are set to present.
3. An unknown signal is set to present, if an emit action is executed.
4. An unknown signal is set to absent, if no emit action can be executed.
5. A transition is enabled, if its trigger is satisfied, and no transition with a higher priority is enabled.
6. A transition cannot be enabled, if its trigger is not satisfied or if a transition with a higher priority is enabled

A syncChart is *constructive* if these rules can be used stepwise, until every signal has a well-defined status. The syncChart `Logical_Correct` is not constructive. Since both A and B are initially unknown and can possibly be emitted (because no transition is ruled out).

3.3 The Simulator

In [4, 3] *André* uses a multi-threaded algorithm to describe the semantics of SyncCharts. A multi-threaded evaluation seems very suitable because of the concurrent nature of SyncCharts, but the complex behavior of SyncCharts will lead to a large programming overhead required to synchronize the computation. It is, for example, not easy to decide if a signal is definitively absent, but it is apparently easy to say when a signal is present. So there are situations where one thread is waiting for a signal to be declared absent and thus delays information used by other threads. The simulation is stuck and an additional thread must be invoked to discover the missing information. That is why this simulator uses a single task and a backtracking mechanism that surely is as complex as the multi-threaded approach but easier to maintain and extend.

The syncChart simulator does not use the KIEL data structure to compute the reactions. It uses its own internal structure that is better adapted to the qualities of SyncCharts. One major advantage of using a separate structure and a conversion routine is that changes during the different development stages and future changes or extensions of the KIEL statechart structure do not affect the simulator. Only adjustments to the conversion routine need to be made.

3.3.1 The Structure

The simulator contains classes to represent simple and macro states. A macro state contains at least one region (also called STG - **S**tate **T**ransition **G**raph). To simplify the computations the behavior of a state is separated from its transitions. That is why an additional layer is introduced: the *cells*. There are *state cells* and *pseudo cells*. A state cell consists of a set of transitions and a state, the cell's body. A pseudo cell contains only transitions. Pseudo cells are used to represent initial states (*initial cells*) and conditional states (*conditional cells*) that do not need to be refined. A STG consists of a set of cells and exactly one initial cell that is used as the entry point. A STG refers to one cell as its *current cell*. The topmost class is the SyncChart that contains exactly one macro state, the top state.

3.3.2 The Results

After the evaluation of an instant, the simulator returns a `MacroStep` (see 4.2.1) object that is further evaluated by the KIEL *browser*. Besides the current status and values of signals and variables, a very important information is the current `Configuration`. A `Configuration` of a syncChart is the set of current active states. Different from Harel's idea of a configuration KIEL configurations in general only contain simple states. If a macro state contains active states, it is also active and the information would be redundant. A macro state can only be a part of the current configuration, if it is suspended and so no one of its child states is active.

The main difference is that KIEL allows configurations to contain pseudo states. *Pseudo configurations* are necessary to display intermediate results of an instant. At the end of an instant a correct syncChart reaches a configuration without pseudo states. Configurations are used as a key to generate and identify the current *view* by the layouter and browser of KIEL. If the configuration changes, the current view might change, too.

3.3.3 The Reaction

The reaction is computed by a recursive algorithm. Every component of the simulator has a `react()` method. The components use return codes to inform their parent elements about their current evaluation status. The evaluation of a parent element depends only on the return code of its child elements. There is no need to know what happened exactly. The most important return codes are:

DEAD :
A final simple state always returns `DEAD`. There is nothing a final simple state can do. A macro state only returns `DEAD`, if all its STGs return `DEAD`. If a macro state returns `DEAD` the surrounding cell might enable its normaltermination transition.

DONE :
A cell returns `DONE`, if it enables a transition. If a cell returns `DONE`, the surrounding STG passes the control to the target cell.

PAUSE :
A cell returns `PAUSE`, if it can not enable a transition. If a cell returns `PAUSE`, the surrounding STG returns `PAUSE`, too. A macro state returns `PAUSE` if at least one of its STGs returned `PAUSE`.

UNKNOWN :
`UNKNOWN` is returned, if a component can not decide, what to do next.

Since the status of all signals is set to `UNKNOWN` at the beginning of every instant, a cell might evaluate a transition with a trigger that can not be evaluated yet. The cell aborts its evaluation and returns `UNKNOWN`. If at least one component returns `UNKNOWN` the evaluation is not finished and the backtracking mechanism will reach every component that returned `UNKNOWN` later again. However, it is possible that there is still not enough information to resume the evaluation. The easiest example would be that a trigger waits for the absence of a signal. Since actions can only be used to set a signals status to present, the simulator needs an additional mechanism to decide, whether a signal is absent.

3.3.4 The Potential

As seen before, the actual emission of signals is not sufficient for the computation to reach a stable configuration. The *absence* of signals is just as important as the *presence* and the current value. If the simulation got stuck, it is obviously a very bad idea to set the status of all not-present signals to absent. This would definitely resume the evaluation, but what should happen, if one of these signals is emitted later during the same instant?

The *potential* at a certain point of the computation is the set of signals that still can be emitted. Every visible signal that is not part of the current potential and that is still unknown can be set to absent. During the computation of the potential count delays and guards are ignored. Only the immediate modifier and signal expressions are regarded. The potential of . . .

a list of actions is the set of all signals that are emitted.

a simplestate is the potential of its oninside actions.

a macrostate is the union of the potential of its oninside actions and the potential of its regions.

a STG is the potential of its current cell.

a transition is the union of the potential of its actions and the potential of its target cell.

a pseudo cell is the potential of its transitions.

a state cell is explained below.

The computation of the statecell starts with its strongabortion transitions. The computation is aborted, if one of the transitions can be enabled. Otherwise the potential of the onentry actions and the body of the cell are computed. The body is omitted if the suspend connector evaluates to true. After that the potential of the weakabortion transitions is computed, and again, if one transitions can be enabled, the computation can be aborted. The potential of the normaltermination transition is only computed, if the evaluation of the body discovers a possible path to a final state in every region. Finally, if at least one transition can be enabled, the potential of the onexit actions of this cell and all cells of the body that were reached during its evaluation are computed. The final result is the union of all potentials.

The computation of a potential allows a large set of syncCharts to be simulated and by the way detects syncCharts that are not constructive. Every syncChart shown in 3.2 is rejected during the simulation, because the potential discovers that the simulation can not be resumed. However, the potential will only identify non-constructive syncCharts, if a problem occurs at run-time.

4 Implementation

The previous chapter contains only a raw overview over the components of the simulator. This chapter describes the implementation in detail using class diagrams and many pseudo code examples.

The components of the simulator are located in ten different packages. To avoid name conflicts between the KIEL data structure and other simulators most of the classes have the name prefix `STRL`. The following sections describe the different packages. All important classes and methods are explained as far as necessary to understand the evaluation.

4.1 The Package kiel.simulator

This is the basic package for all simulators used in KIEL. The different simulators are located in different subpackages. The syncChart simulator is located in `kiel.simulator.syncchart`.

4.1.1 The Class SimulatorException

This is the basic class for all exceptions that occur during the simulation process. Section 4.4 describes the different exceptions handled by the syncChart simulator.

4.1.2 The Interface Simulator

The interface `Simulator` must be implemented by all simulators used in KIEL.

setStateChart(StateChart) :
This method sets the KIEL statechart that is going to be simulated.

emit(Event) :
This method must be called for each pure input signal that should be present during the next instant. Within the KIEL data structure the basic class is called `Event` instead of `Signal` because of the UML basis.

emit(Event,int) :
This is the corresponding method for valued signals.

nextstep() :
After all input signals are set, this method is called to compute the instant.

The result is a `MacroStep` object (see 4.2.1) that now can be evaluated by the KIEL browser.

reset() :
Resets the simulation, so that the next call of `nextstep()` starts with the first instant again.

4.2 The Package kiel.configMngr

The package `kiel.configMngr` contains all classes to represent the results of the simulation. This package is not part of the simulator package hierarchy because parts of it are used by other KIEL modules that have nothing to do with the simulation.

Figure 4.1 shows the different classes that are used to describe what happens during one instant. `MicroStep` is an interface for four different classes that describe the current status of signals, states, transitions and values.

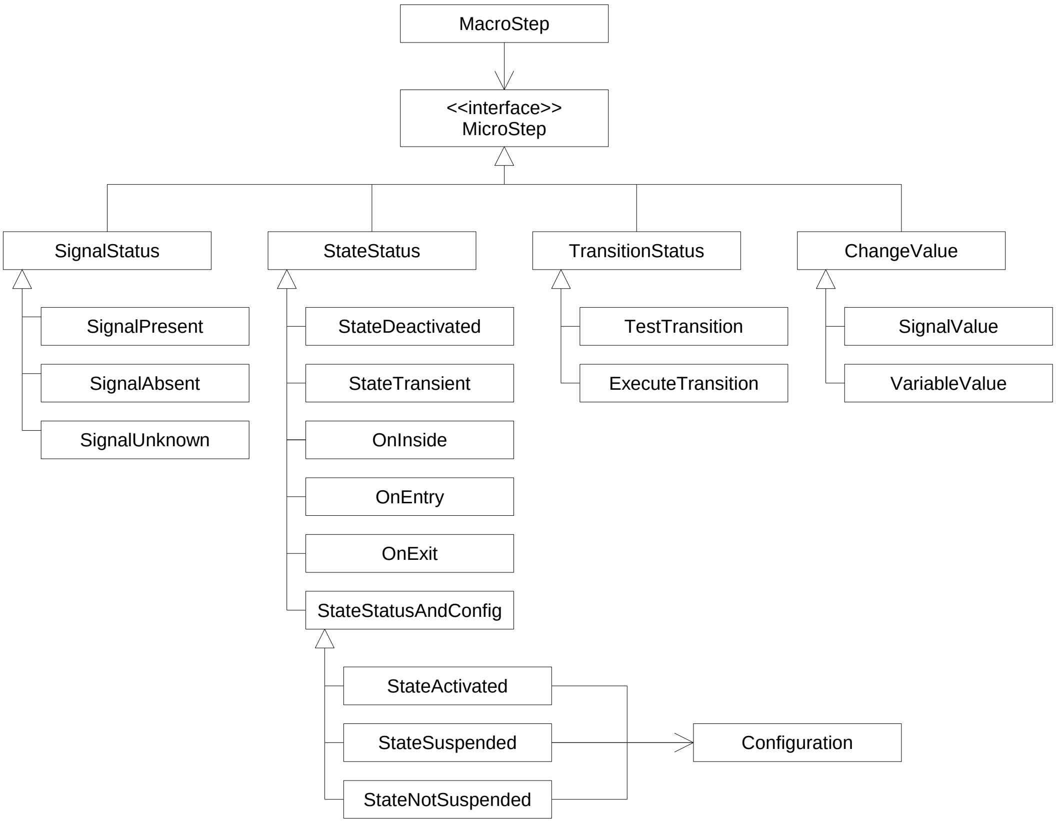


Figure 4.1: Type hierarchy showing the different microsteps.

4.2.1 MicroSteps and MacroSteps

A syncChart evolves in discrete instants. During each instant, signal statuses, active states and values might change. The class MacroStep contains nothing but a list of MicroSteps that describe everything what happens in the instant and might be interesting to view in the browser. The order of the microsteps within the macrostep represents the order they occur during the evaluation. Although everything is said to happen simultaneously this order implies a causality.

The Class SignalStatus

The class SignalStatus contains a reference to a signal that changes its current status. The subclasses SignalPresent, SignalAbsent and SignalUnknown are used to specify the status. A SignalAbsent object for example is inserted in the list of microsteps at the moment the simulator discovers that no emit action for this signal is reachable anymore in the same instant.

The Class StateStatus

The basic class StateStatus is used for everything that involves states. Each subclass has a reference to the corresponding state. OnEntry, OnInside and OnExit objects are generated, if a state executes its onentry, oninside or onexit actions respectively. StateTransient is generated if a state becomes transient due to an immediate strongabortion. StateDeactivated is generated if a state is not transient and enables an outgoing transition.

StateStatusAndConfig is the basic class for the remaining three subclasses. They not only contain a reference to a state, but also a new configuration. New configurations are generated if the set of active states changes. So StateActivated is generated if a state is reached by an incoming transition and not jumped over with an immediate strongabort. The suspend connector might also influence the set of active states. This circumstance is included with StateSuspended and StateNotSuspened. The latter three microsteps might trigger the browser to receive a new view (possibly containing a new layout) from the layouter.

The Class TransitionStatus

For transitions there are only two events worth mentioning. TestTransition is generated if the trigger of a transition is tested. Due to the backtracking mechanism a transition can be tested several times until it is determinable whether the trigger is satisfied or not. ExecuteTransition is generated if a transition is taken.

The Class ChangeValue

These classes are generated if a signal or variable changes its current value. For single signals a SignalValue is generated right at the moment the signal is emitted. For combine signals the microstep can not be generated until the simulator is sure that the signal is not emitted any more during this instant. Contrary to signals variables can change their values several times within one instant, so multiple `VariableValue` objects can refer to the same variable and thus replace the value of their predecessor.

4.3 The Package kiel.simulator.syncchart.converter

This package contains 21 classes used to transform a KIEL statechart into the internal representation. The algorithm is straightforward because of the simple tree-like structure. Each class transforms a specific component of the original chart and uses recursive calls of other converter classes to transform the child elements.

It is assumed that the KIEL statechart is correct and contains only components that are compatible with the syncChart simulator. Anyway exceptions are thrown if for example a region does not have an initialstate

The only class used by the syncChart simulator is `KielToSyncchart`. The method `SyncChart convert(StateChart s)` takes a kiel `StateChart` and returns an internal `SyncChart`.

4.4 The Package kiel.simulator.syncchart.exceptions

This package contains different types of exceptions that can occur during the conversion process or the simulation itself. Actually, it is assumed that the KIEL statecharts are syntactically and semantically correct. That is why the exception concept is not very complex. It was very helpful though during the early development stages of both the simulator and the other KIEL modules.

STRLException :
This is the generic basic exception. It is thrown if for example the simulator can not derive the status of a signal after the evaluation got stuck. This happens when the syncChart is not constructive (see 3.2).

STRLUninitializedReadException :
This exception is thrown, if a valued signal or variable is defined without a default value and there is a read access before a value is assigned.

STRLSignalNotFoundException :
This exception is thrown, if a signal is used as part of a trigger or an action, but not defined in the correct scope.

STRLIntegerSignalNotFoundException :
The same as above, but for valued signals.

STRLVariableNotFoundException :
The same as above, but for variables.

STRLNotSupportedException :
This exception is thrown during the conversion process, if the statechart contains an object that is not part of the SyncCharts syntax.

4.5 The Package kiel.simulator.syncchart.intexp

This package contains classes to construct integer expressions. As seen in figure 4.2 all classes implement the `STRLIntegerExpression` interface. The common interface allows to construct almost arbitrary expressions.

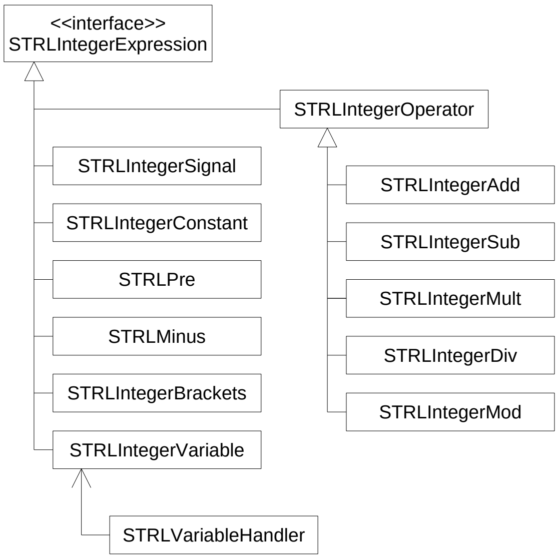


Figure 4.2: Type hierarchy of the integer expression package.

4.5.1 The Interface STRLIntegerExpression

The interface `STRLIntegerExpression` contains only three methods:

getValue() :
The expression is evaluated using the current values of all used signals and variables and the value is returned.

ready() :
`ready()` returns true if and only if every used signal or variable is *ready*. This is explained in sections 4.7 and 4.8. If an expression is not *ready* `getValue()` will return a wrong value or even throw an exception.

getUsedVariables() :

Returns the set off all variables used within this expression. This is necessary for the execution of actions (see 4.8).

Using the usual tree structure, all methods concerned with integer expressions are evaluated recursively. For example `STRLIntegerAdd` consists of two other integer expressions and so the call of `ready()` returns true if both of the summands return true. Integer expressions are used in assignments, valued emit actions, valued signal and variable declarations, guards and counting delays.

4.5.2 The Class STRLVariableHandler

Each region uses a `STRLVariableHandler` to manage its local variables. The handler is used to initialize the variables if the region is entered.

4.6 The Package kiel.simulator.syncchart.boolexp

Figure 4.3 contains the classes of the Boolean expression package. The classes on the left half, like `STRLAnd` or `STRLPre`, are used to construct arbitrary Boolean expressions with signals used as signal expressions to trigger transitions. The right half contains the different `STRLComparator` classes. Each comparator contains two integer expression that are compared with the corresponding operator if the expression is evaluated. Boolean expressions used as guards are not allowed to contain any signal statuses.

Boolean expressions are evaluated with the method `evaluate()`. This method might return three different constants: `FALSE` (`ABSENT`), `TRUE` (`PRESENT`) or `UNKNOWN`. `UNKNOWN` is used if the Boolean expression can not be evaluated at the moment, because a signal status is `UNKNOWN` or an integer expression is not *ready*. Figure 4.4 shows how `STRLAnd`, `STRLOr` and `STRLNot` are evaluated depending on the evaluation of their child expressions.

4.6.1 The Class STRLPre

The evaluation of `STRLPre` is not obvious for arbitrary signal expressions. There is a problem if `pre` is applied to a signal that was not active during the previous instant and thus does not have a valid status in the previous instant. That is why signals use two helping methods that deal with the initial status.

`pre_0()` :

Returns the previous status of the signal or `ABSENT` if the signal has no previous status.

`pre_1()` :

Returns the previous status of the signal or `PRESENT` if the signal has no previous status.

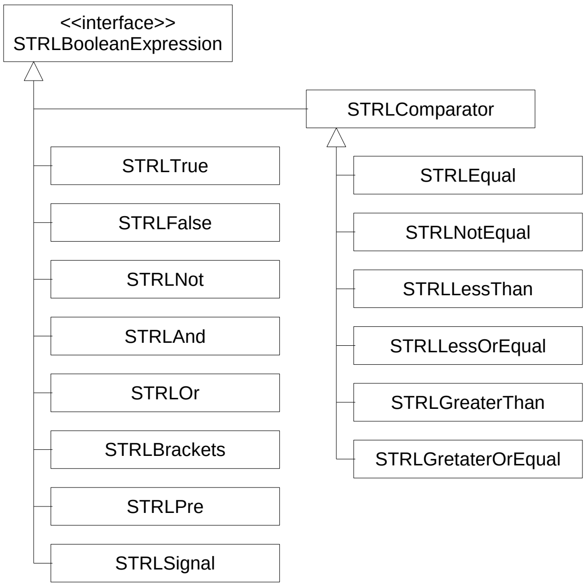


Figure 4.3: Type hierarchy of the Boolean expression package.

STRLAnd

left \ right	true	false	unknown
true	true	false	unknown
false	false	false	false
unknown	unknown	false	unknown

STRLOr

left \ right	true	false	unknown
true	true	true	true
false	true	false	unknown
unknown	true	unknown	unknown

STRLNot

	not
true	false
false	true
unknown	unknown

Figure 4.4: Evaluation of and, or and not using true, false and unknown.

Now the previous operator can be defined by structural induction for arbitrary signal expression `e`, `e1` and `e2`.

```
pre( e ) = pre_0( e )

pre_0( e1 and e2 ) = pre_0( e1 ) and pre_0( e2 )

pre_1( e1 and e2 ) = pre_1( e1 ) and pre_1( e2 )

pre_0( e1 or e2 ) = pre_0( e1 ) or pre_0( e2 )

pre_1( e1 or e2 ) = pre_1( e1 ) or pre_1( e2 )

pre_0(not e) = not pre_1( e )

pre_1(not e) = not pre_0( e )
```

The benefit from this definition is that both `pre( S )` and `pre( not S )` return `ABSENT`, if the signal `S` has no previous status.

4.7 The Package kiel.simulator.syncchart.sigexp

The package `sigexp` contains all classes concerning signals. Figure 4.5 contains an overview of all the signal classes.

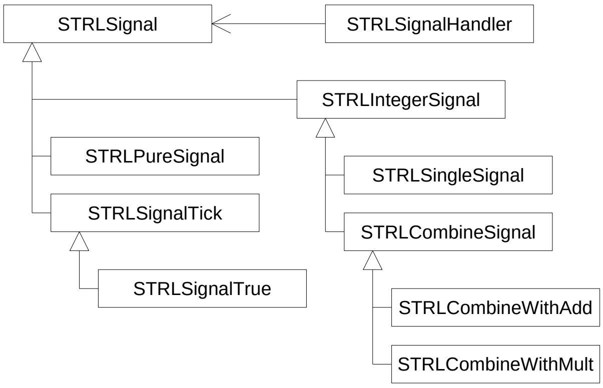


Figure 4.5: Type hierarchy of the different signal types.

4.7.1 The Class STRLSignal

`STRLSignal` is an abstract class used as basis for all signals. `STRLSignal` contains private fields to store the `currentStatus` and the `previousStatus` and several methods to access and change these values. Every change of the current status will generate a `SignalStatus` microstep that is appended to the current microstep list.

4.7.2 The Class STRLPureSignal

`STRLPureSignal` inherits all its methods from `STRLSignal` and adds:

`initializeStatus()` :
Resets both the current and the previous value to `UNKNOWN`. This method is called if the scope of this signal is entered.

`resetStatus()` :
This method is called if the scope of the signal becomes active again. The *old* current status is stored in the previous status and the current status is set to `UNKNOWN`.

`emit()` :
Sets the current status to `PRESENT`.

`update(Collection signals)` :
This sets the status of the signal to `ABSENT` if the current status is still `UNKNOWN` and the given collection does not contain this signal.

4.7.3 The Class STRLSignalTick

This is the predefined signal `tick` and this class does nothing but return `PRESENT` when evaluated.

4.7.4 The Class STRLSignalTrue

This class also always returns `PRESENT` when evaluated. This class is used for empty transition triggers.

4.7.5 The Class STRLIntegerSignal

`STRLIntegerSignal` is an abstract basic class for all valued signals. In addition to the status, a valued signal also stores a current and previous value. A STRLIntegerSignal is constructed with an optional `STRLIntegerExpression` that is evaluated as a default value, when the scope of the signal is reached. `STRLIntegerSignal` contains a Boolean flag `ready` that is true if the value of the signal is readable.

The Class STRLSingleSignal

The `emitValue(int value)` method sets the current value to the given value and sets the `ready` flag to `true`. The method `ready()` inherited from `STRLIntegerExpression` will now return true, and the new value is accessible. If the signal is emitted twice within one instant an exception is thrown.

The Class STRLCombineSignal

`STRLCombineSignal` is an abstract class as basis for `STRLCombineWithAdd` and `STRLCombineWithMult`. The main difference is that combine signals can be emitted arbitrary times within one instant. The current value is accumulated using the appropriate combining function. A combine signal sets its `ready` flag to `true` only if the simulator discovers that no emit value action for this signal is reached anymore.

4.7.6 The Class STRLSignalHandler

Each macro state uses a `STRLSignalHandler` to manage its local signals. The handler can initialize the signals, if the state is newly entered, or reset the signals, if the state was already active in the previous instant.

4.8 The Package kiel.simulator.syncchart.action

This package contains everything concerning actions. Actions are not only used to emit pure and valued signals, to assign new values to variables but also to initialize signals, variables and transitions. Initializing actions are usually executed if the scope of the object is reached.

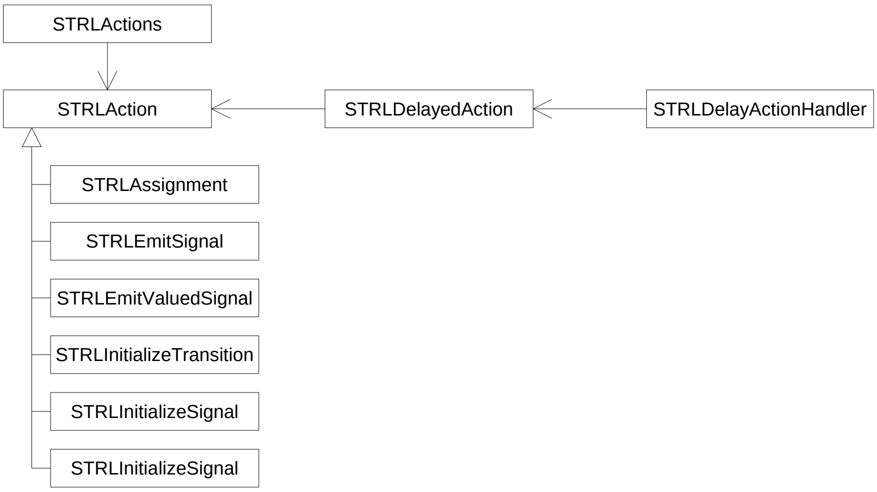


Figure 4.6: The type hierarchy of the action package

Figure 4.6 shows the different action classes.

4.8.1 The Class STRLAction

`STRLAction` is the abstract basic class for all actions. Each action must implement the following methods:

`execute()` :
: This executes the action.

`ready()` :
: This returns true, if the action can be executed. Since all actions are involved with integer expression, the call to `ready()` is passed the the particular integer expression.

`getVariablesRead()` :
: Returns a list of all variables that are read to execute this action.

`getVariableWritten()` :
: Returns the variable that is written or `NULL` if the action is not an `STRLAssignment`.

`getPotential()` :
: This returns a set containing the signal that is emitted or an empty set for other actions. The *potential* concept 3.3.4 is used to derive which signals are `ABSENT`, if the simulation freezes.

The Class STRLAssigment

`STRLAssignment` is used to assign the value of an integer expression to a variable.

The Class STRLEmitSignal

`STRLEmitSignal` is used to emit a pure signal. Valued signals can not be emitted without a value.

The Class STRLEmitValuedSignal

`STRLEmitValuedSignal` is used to emit either a single signal or a combine signal with the value of an integer expression.

The Class STRLInitializeSignal

If a macro state is reached and its body becomes active, the `STRLInitializeSignal` action is used to assign the default value to all local signals.

The Class STRLInitializeVariable

Variables are initialized in the same way as signals.

The Class STRLInitializeTransition

Transitions need to be initialized, if they have an integer expression used as a count delay in the trigger part. If a state is reached its strongabortion transitions need to be initialized at first. The weakabortion transitions are not initialized until the reaction of the states body has finished.

The Class STRLActions

The class `STRLActions` merges different actions into one list. States and transitions have only references to `STRLActions` objects and not to many `STRLAction` objects.

4.8.2 Delayed Action Handling

Every time an action is reached the possibility exists that the action can not be executed yet, because it depends for example on another signal that is not *ready*. A single signal is said to be *ready*, if it was emitted during this instant, or if the simulator discovers that the signal will not be emitted at all. A combine signal is only said to be *ready*, if the simulator discovers that it will not be emitted any more during this instant. As long as a signal is not ready, its value can not be accessed.

If an action can not be executed immediately it becomes wrapped with a `STRLDelayedAction` and is then handled by the `STRLDelayedActionhandler`. The execution of an action does not effect the state or transition this action was assigned to.

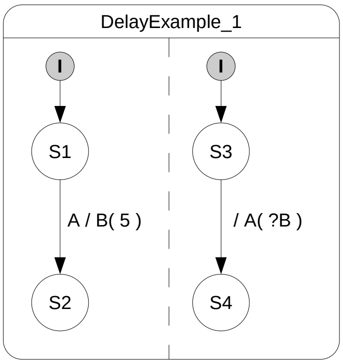


Figure 4.7: Delayed action handling

Figure 4.7 can be used to show how the delayed action handling works. In the first instant the states *S1* and *S2* become active. The simulator uses a single thread so the second instant might start with the left region. The transition can not be enabled because the status of the signal *A* is still unknown. The evaluation of the left region is aborted and the right region is evaluated. The transition is enabled and the signal *A* should be emitted with the value of the signal *B*. Since *B* is not ready, the value can not be accessed. The emission of A becomes *delayed* but the transition is taken anyway. For valued signals the emission of the status and value are separated. So A is now present and the evaluation of the right region is finished in state *S4*. Now that A is present the evaluation of the left region is resumed and the transition is taken. *S2* becomes active and B is emitted with the value 5. Now the evaluation of the syncChart is finished, but there is still one delayed action left. Since B is ready now A can finally receive the value.

The Class STRLDelayedActionHandler

The `STRLDelayedActionHandler` handles action and constructs delayed action. Every action that can not be executed immediately is inserted in the set of delayed action. Every time the simulation got stuck the handler tries to execute the delayed actions again.

The Class STRLDelayedAction

To simplify the decision whether an action can be executed or must be delayed, every action that is reached is immediately wrapped by a *STRLDelayedAction*. The constructor of the delayed action constructs a set of references to other delayed actions that this action depends on. So a delayed action can only be executed if its set of depending actions is empty. As seen before, actions depend on other actions if signals are not ready. However, not ready signals might also delay the assignments to a variable. Every action reached after this assignment might depend on that variable. That is what the `getUsedVariables()`, `getVariablesRead()` and `getVariablesWritten()` methods are for.

So if a delayed action is constructed it depends on every other delayed action that

1. writes a variable that is read in this action.

2. uses a variable that is written by this action.

3. emits a signal with a value that is read in this action.

Figure 4.8 shows a more complicated example that is easily handled with delayed actions.

4.9 The Package kiel.simulator.syncchart.transition

Figure 4.9 shows the classes of the transition package.

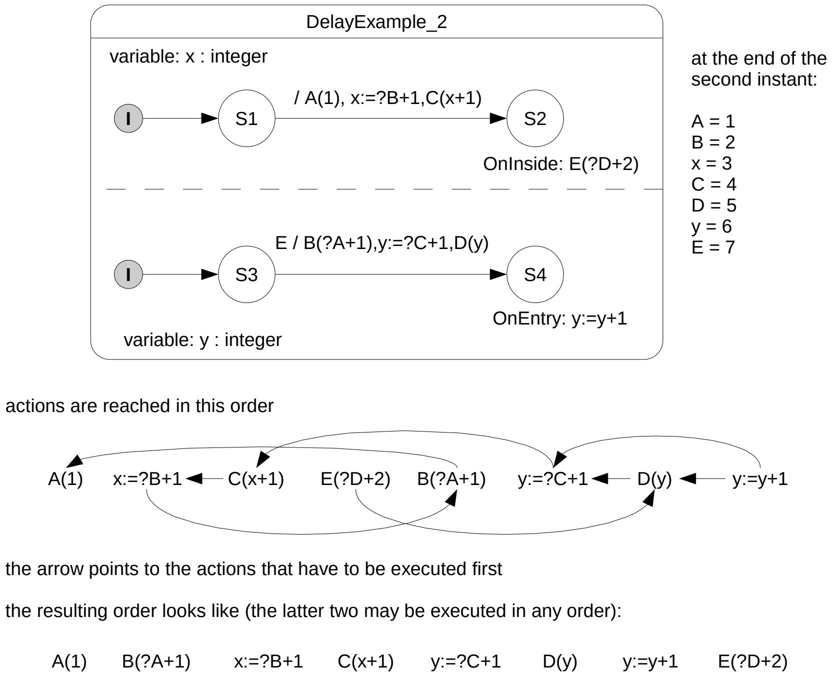


Figure 4.8: Interdependence of different regions is solved with delayed action handling

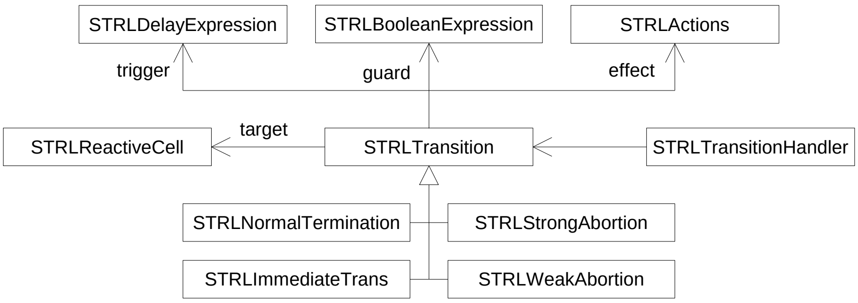


Figure 4.9: Classes of the transition package

4.9.1 The Class STRLDelayExpression

The class `STRLDelayExpression` is used to combine the immediate modifier, the count delay and the signal expression (immediate modifier and count delay are still exclusive). The `STRLDelayExpression` counts how often the signal expression was satisfied and compares this with the actual value of the count delay. A delay expression is evaluated with a call to `evaluate(boolean firstInstant)`. The return value is as usual one of the following constants: `NOTSATISFIED`, `SATISFIED` or `UNKNOWN`. The Boolean flag `firstInstant` indicates whether this is the first instant for the source state. At the first instant all delay expression without the immediate flag will return `NOTSATISFIED` without any further computations. As usual, the return value is `UNKNOWN` if the signal expression can not be evaluated.

4.9.2 The Class STRLTransition

`STRLTransition` is the abstract basic class for the different transition types. It contains the following fields and methods:

`STRLDelayExpression delayExp` :
: The trigger of the transition.

`STRLBooleanExpression guard` :
: The Boolean guard is evaluated only after the trigger returned `SATISFIED`.

`STRLActions effect` :
: The list of actions is executed if both the trigger and the guard are satisfied.

`STRLReactiveCell target` :
: The target is a cell instead of a state. The *cell* concept is used to separate the state and its body from its outgoing transitions. This is explained in section 4.10.

`boolean entersHistory` :
: This flag is true, if the target state is entered through a history connector.

`evaluate(boolean firstInstant)` :
: This method is used to evaluate the trigger and guard of the transition. There are three return constants: `ENABLED`, `NOTENABLED` and `UNKNOWN`.

`execute()` :
: This finally executes the actions of the transition.

The subclasses `STRLStrongAbortion`, `STRLWeakAbortion`, `STRLNormalTermination` and `STRLImmediateTrans` are used for the different transition types.

4.9.3 The Class STRLTransitionHandler

The `STRLTransitionHandler` is used to handle outgoing transitions of the same type. A macro state (more precisely its surrounding *cell*) uses three transition handlers for its outgoing transitions.

A `STRLTransitionHandler` provides the following methods:

`init()` :
: This method is used to initialize the count delay of the transitions.

`evaluate(boolean firstInstant)` :
: This evaluates the transitions. Starting with the transition with the highest priority until a transition is reached that evaluates to `SATISFIED` or `UNKNOWN`. If the result is `UNKNOWN` the evaluation of this handler/state/region is aborted and later resumed with the evaluation of the same transition. If the result is `SATISFIED`, the evaluation is stopped and the transition can be taken.

`execute()` :
: This is called after the handler has found a satisfied transition.

The `evaluate` method of the transition handler returns one of three return codes:

- `FALSE`: No transition can be enabled.

- `TRUE`: A transition can be enabled.

- `UNKNOWN`: It is unknown whether a transition can be enabled or not.

If the transition handler executes a transition, it also prepares the target state, so that the targets reaction can be computed next.

4.10 The Package kiel.simulator.syncchart

The package `syncchart` contains the main classes that represent the internal syncChart structure and are used to compute the reactions. Figure 4.10 shows a raw type overview of the classes in this package and figure 4.11 shows a detailed view of the interaction and association between the classes of the former packages. The cells are used to separate the evaluation of the outgoing transitions from the source state itself. The name originates from the look of the states and transitions compared to (nerve) cells and dendrites. The idea is that the reaction is computed by cells that communicate via dendrites with each other.

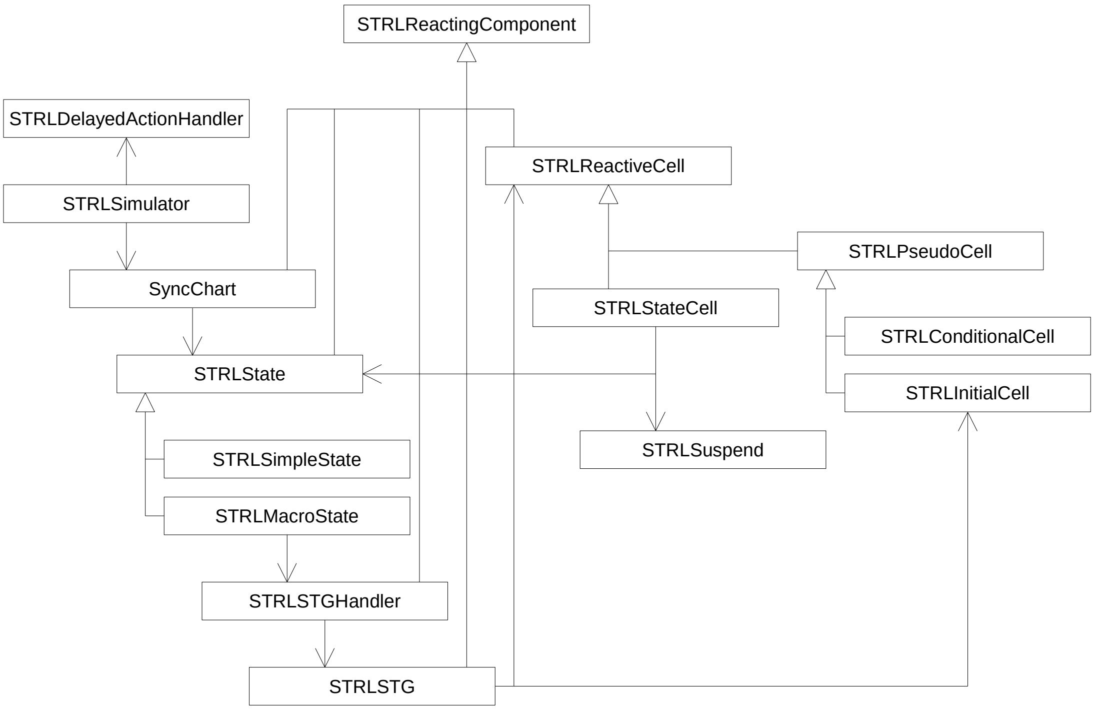


Figure 4.10: Overview of the most important classes in the package: syncchart.

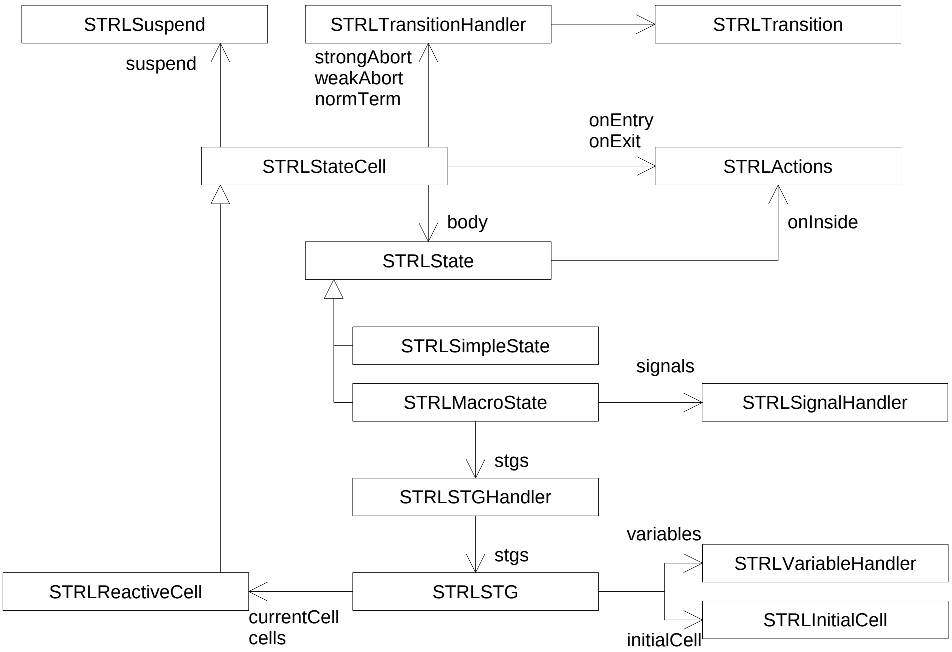


Figure 4.11: Important associations between classes of different packages

4.10.1 The Class STRLReactingComponent

`STRLReactingComponent` is the basic class whose methods must be implemented by all classes that are involved in the reaction. The most important methods are:

`getActiveStates()` :
: This method works itself recursively through the hierarchy tree and finally returns the set of active states (the configuration).

`prepareForNext()` :
: Many components use flags that need to be reset before the next evaluation. A `STRLStateCell` contains the flag `firstInstant` that is true, if it is the first instant for the corresponding state. This flag must be set to false, if the next instant is evaluated.

`deepEnter(boolean throughHistory)` :
: This method is called recursively, if a macro state is entered, to inform all contained components. The flag `throughHistory` tells whether the state was entered with a history connector or not. If the flag is true, variables, signals and count delays are not reset.

`deepExit()` :
: If a state is exited, this method is used to execute the onexit actions of its child states.

`react()` :
: This is the most important method to compute the reaction. The `react()` method can return four different constants: `DEAD`, `DONE`, `PAUSE` or `UNKNOWN`.

`update(Collection signals)` :
: This method is used to set currently unknown signals to absent, if no emit action is reachable anymore.

4.10.2 The Class STRLSimulator

The most important method of the `STRLSimulator` is `nextStep()`, which is called by the browser. The following listing shows a simplified version.

```
MacroStep nextStep(){
    int code = UNKNOWN;
    while (code == UNKNOWN) {
        this.somethingchanged = false;
        code = this.syncChart.react();
        this.handler.execute();
        Collection potential = this.syncChart.getPotential;
        potential.addAll(this.handler.getPotential());
        syncChart.update(potential);
        if (code == UNKNOWN && !this.somethingchanged) {
```

```
11            throw this.createException(potential);
12         }
13      }
14      boolean test = false;
15      while (!test) {
16         this.somethingchanged = false;
17         test = this.handler.execute();
18         Collection potential = this.handler.getPotential();
19         this.syncChart.update(potential);
20         if (!this.somethingchanged && !test) {
21            throw this.createException(potential);
22         }
23      }
24      this.instant++;
25      MacroStep result = this.macroStep;
26      this.macroStep = new MacroStep(this.instant);
27      this.syncChart.prepareForNext();
28      return result;
29   }
```

lines 3 - 13 :

As long as the syncChart returns `UNKNOWN` the reaction has not finished. To avoid an infinite loop a flag `somethingchanged` is set to false at the beginning of every iteration. This flag is true, if a signal changes its status or value, a variable changes its value or a state changes its status (active, suspended).

line 6 :

The delayed action handler tries to execute the delayed actions.

line 7 :

The syncChart returns its potential. The potential is the set of signals that still can be emitted (see 3.3.4).

line 8 :

The potential of the delayed action handler is added.

line 9 :

All signals that are visible in the current scope are updated. Every signal that has still an unknown status is set to absent, if not contained in the potential.

line 10 :

If the syncChart returns `UNKNOWN`, but nothing has changed, nothing would change in the next iteration. The syncChart is not constructive, an exception is thrown, the simulation is aborted.

line 14 - 23 :

After the syncChart returns a different code (the reaction has finished), there can still be some delayed actions. And if these can not be executed completely, the syncChart is not constructive.

line 17 :
The execute method of the delayed action handler returns true, if all actions have been executed.

line 24 - 28 :
The reaction has finished, all actions have been executed. The `MacroStep` object in line 25 was global accessible and contains every microstep that was generated during the evaluation.

4.10.3 The Class STRLState

`STRLState` is a basic class that contains various fields and get/set methods common to both simple states and macro states.

4.10.4 The Class STRLSimpleState

The reaction of a simple state is very easy.

```
 1  int react() {
 2     if (this.isFinal()) {
 3        return DEAD;
 4     } else {
 5        if (!this.onInsideExecuted) {
 6           if (!this.getOnInside().isEmpty()) {
 7              this.getOnInside().execute();
 8           }
 9           this.onInsideExecuted = true;
10        }
11        return PAUSE;
12     }
13  }
```

line 2 - 4 :
The syncChart structure does not contain special final states. Each state has a flag instead that is true, if the state is final. So if the state is a final state, it returns `DEAD`. Final states do not have oninside actions.

line 5 - 13 :
At first it is checked, if the oninside actions have not been executed yet. This is necessary because of the iterated call of the `react()` method. The actions are executed, if they are not empty, the flag is set to true, and `PAUSE` is returned.

4.10.5 The Class STRLMacroState

The reaction of a `STRLMacroState` is still quite easy. A macro state uses a status value to store its last action. The status value is used to resume the reaction where it was aborted.

```
 1 int react() {
 2    if (!this.onInsideExecuted) {
 3       if (!this.getOnInside().isEmpty()) {
 4          this.getOnInside().execute();
 5       }
 6       this.onInsideExecuted = true;
 7    }
 8    switch (this.position) {
 9    case (INITSIGNALS):
10       return this.initSignals();
11    case (RESETSIGNALSTATE):
12       return this.resetSignalStatus();
13    default:
14       return this.reacting();
15    }
16 }
17
18 int reacting() {
19    if (this.lastResultOfSTGs == UNKNOWN) {
20       this.lastResultOfSTGs = this.stgs.react();
21    }
22    return this.lastResultOfSTGs;
23 }
```

line 2 - 7 :
A macro state executes its oninside actions like a simple state.

line 8 :
The value `position` is used to select the right method. If the macro state is entered for the first time or reentered without a history connector the value is set to `INITSIGNALS`. If the macro state is still active at the end of an instant, the value is set to `RESETSIGNALSTATUS`.

line 10 :
The method `initSignals()` executes the `STRLInitializeSignal` actions for the local signals. After that, the method `reacting()` is called.

line 12 :
The method `resetSignalStatus()` prepares the local signals. The different reset methods store the old current status and values in the previous status and value and set the new current status to `UNKNOWN`. After that, the method `reacting()` is called.

line 18 - 23 :
The method `reacting` passes the call to the different regions of the macro state. The class for a region is called `STRLSTG` (stg - state transition graph). The `STRLSTGHandler` handles the reaction of the stgs. The macro state stores the last return value. If the value is not `UNKNOWN`, the regions either returned

DEAD or PAUSE and have nothing left to to. The macro state returns the code of its regions.

4.10.6 The Class STRLSTG

The class STRLSTG represents a region. An stg contains a set of reactive cells, one STRLInitialCell and one reference to the currentCell. If the region is entered or reentered without a history connector, the currentCell points to the initial cell.

Like the macro state, the stg needs a position value to resume the reaction at the right point.

```
 1  int react() {
 2     if (this.lastResult == UNKNOWN) {
 3        if (this.position == INITIALIZING) {
 4           return this.initializing();
 5        } else {
 6           return this.executing();
 7        }
 8     } else {
 9        return this.lastResult;
10     }
11  }
12
13  int executing() {
14     if (this.lastResult == UNKNOWN) {
15        do {
16           this.lastResult = this.currentCell.react();
17           if (this.lastResult == DONE) {
18              this.currentCell = this.currentCell.nextCell();
19           }
20        } while (this.lastResult == STRLReactingComponent.DONE);
21     }
22     return this.lastResult;
23  }
```

line 2 :
The stg stores the last value of its reaction to avoid redundant computations.

line 3 :
position is either INITIALIZING or REACTING. If the region is entered or reentered without a history connector, position is set to INITIALIZING.

line 4 :
The method initializing() is used to execute the STRLInitializeVariable actions for the local variables. After that the method executing() is called.

line 13 - 23 :
The stg reacts as long as the current cell returns DONE. Actually the current cell is allowed to change arbitrary times (except for infinite immediate loops).

If a cell returns DONE, it has enabled and already executed a transition. The method nextCell() returns the target of that transition. If the cell returns UNKNOWN, the react() method will be called again later. If the cell returns DEAD or PAUSE, the value is returned.

4.10.7 The Class STRLSTGHandler

The STRLSTGHandler is used to handle the regions/stgs for a macro state. The call of react() is simply passed to all stgs.

```
int react() {
   int maximum = DEAD;
   Iterator iter = this.stgs.iterator();
   while (iter.hasNext()) {
      STRLSTG next = (STRLSTG) iter.next();
      int temp = next.react();
      if (temp > maximum) {
         maximum = temp;
      }
   }
   return maximum;
}
```

Every stg is allowed to react once, and the maximum of the return codes is returned. It is essential that DEAD < DONE < PAUSE < UNKNOWN. So if all stgs return DEAD the surrounding macro state will also return dead and thus activate its normal-termination transition. If no stg returns UNKNOWN but at least one return PAUSE the macro state will return PAUSE, too. If at least one stg return UNKNOWN, this method is called again later.

4.10.8 The Class STRLSuspend

The STRLSuspend is used by the STRLStateCell to decide whether the body of the cell should be suspended or not. STRLSuspend contains a signal expression and an immediate flag. The evaluate(boolean firstInstant) works like the evaluation of transition triggers.

4.10.9 The Class STRLReactiveCell

STRLReactiveCell is the small basic class for all cells, so that the stg does not need to distinguish between normal states and pseudo states.

4.10.10 The Class STRLPseudoCell

STRLPseudoCell is a basic class for STRLIntialCell and STRLConditionalCell. A pseudocell only contains a transition handler with at least one immediate transition.

```
1  int react() {
2     int code = this.handler.evaluate();
3     if (code == FALSE) {
4        throw new STRLException("...");
5     } else if (code == UNKNOWN) {
6        return UNKNOWN;
7     } else {
8        this.setNextCell(this.handler.nextCell());
9        this.handler.execute();
10       return DONE;
11    }
12 }
```

line 4 :

If the handler returns `FALSE` the pseudo state can not be left in this instant. The default transition is missing.

line 7 - 10 :

If a transition can be enabled, a reference to the target cell is stored for the surrounding stg. The actions of the transition are executed and `DONE` is returned. The surrounding stg will continue with the target cell.

There is no difference between the behavior of a conditional cell and an initial cell.

4.10.11 The Class STRLStateCell

`STRLStateCell` is the largest class (700 lines, including javadoc). Most of the reaction of a syncChart takes place in state cells. The react method contains a large switch-statement.

```
int react() {
   switch (this.position) {
   case RESETSA:
      return this.resetSA();
   case CHECKSA:
      return this.checkSA();
   case CHECKENTRY:
      return this.checkEntry();
   case CHECKSUSP:
      return this.checkSusp();
   case EXECUTEBODY:
      return this.executeBody();
   case STRLStateCell.CHECKWA:
      return this.checkWA();
   case STRLStateCell.CHECKNT:
      return this.checkNT();
   case STRLStateCell.EXIT:
      return this.exit();
   }
}
```

There are eight different position codes, but the reaction can only be aborted and later resumed at CHECKSA, CHECKSUSP, EXECUTEBODY and CHECKWA. Only the signal testing can freeze the reaction.

A state cell always starts its reaction with RESETSA.

```
int resetSA() {
   if (this.isFirstInstant) {
      this.strongAbort.init();
   }
   this.position = CHECKSA;
   return this.checkSA();
}
```

The strongabortion transitions are only reset, if it is the first instant of this state cell. Finally the method checkSA() is called.

```
int checkSA(){
   int check = this.strongAbort.evaluate(this.isFirstInstant);
   if (check == UNKNOWN) {
      return UNKNOWN;
   } else if (check == FALSE) {
      this.position = CHECKENTRY;
      return this.checkEntry();
   } else {
      if (this.isFirstInstant) {
         this.isTransient = true;
      }
      this.exitHandler = this.strongAbort;
      this.position = EXIT;
      return this.exit();
   }
}
```

At first the transition handler for the strongabortion transitions is evaluated. If the handler return UNKNOWN the state cell also returns UNKNOWN. If react() is called again later, the evaluation will continue with the strongabortion transitions. If no transitions is enabled, the method checkEntry() is called. If a transition is enabled, the state cell might become flagged *transient*, if this is the first instant of this cell. The strongabortion handler is assigned to the exitHandler, so that it can be used in the exit() method. The exit handler is used, so that the exit() method does not need to know, what type of transition was enabled.

```
int checkEntry() {
   if (this.isFirstInstant) {
      if (!this.onEntry.isEmpty()) {
         this.onEntry.execute();
      }
   }
   this.position = CHECKSUSP;
   return this.checkSusp();
}
```

The onentry actions are only executed, if it is the first instant of this cell. After that, the state cell checks the suspend connector.

```
checkSusp() {
   int check = this.suspend.evaluate(this.isFirstInstant);
   if (check == TRUE) {
      this.bodyCode = PAUSE;
      this.isSuspended = true;
      return this.resetWA();
   } else if (check == FALSE) {
      if (this.isSuspended) {
         this.isSuspended = false;
      }
      this.position = EXECUTEBODY;
      return this.executeBody();
   } else {
      return UNKNOWN;
   }
}
```

If the suspend connector evaluates to true, the evaluation of the body of the cell is overridden by manually setting the `bodyCode` to `PAUSE`. The next method is `resetWA()`, which resets and then checks the weakabortion transitions equivalent to `resetSA()` and `checkSA()`. If the suspend connector returns false (every state cell has a default connector that always returns false), the body is finally executed with `executeBody()`. If the flag `isSuspended` is true, it is now set to false. If the flag was true, the body was suspended during the last instant.

```
int executeBody() {
   this.bodyCode = this.body.react();
   if (this.bodyCode == UNKNOWN) {
      return this.bodyCode;
   } else {
      return this.resetWA();
   }
}
```

This method finally closes the circle of recursive calls of `react()`. The body of a state cell is a either a macro state or a simple state. Unless the body returns `UNKNOWN`, the evaluation continues with the weakabortion transitions.

The handling of the weakabortion transitions is exactly like the handling of the strongabortion transitions. If no weakabortion transition can be enabled, the normaltermination transition is checked.

```
int checkNT() {
   if (this.bodyCode == DEAD) {
      int check = this.normTerm.evaluate(false);
      if (check == STRLTransitionHandler.TRUE) {
         this.exitHandler = this.normTerm;
         this.position = EXIT;
         return this.exit();
      } else if (check == FALSE) {
            if (this.body.isFinal()) {
                return DEAD;
            } else {
                return PAUSE;
            }
        } else {
            return UNKNOWN;
        }
    } else {
        return PAUSE;
    }
}
```

The normaltermination transition is only evaluated, if the body returned `DEAD`. The evaluation looks similar to the evaluation of other transitions, but a state can only have one normaltermination transition, and this transition has no trigger or guard. So the handler will always evaluate to `TRUE` of `FALSE`, depending on the presence of such transition. If no transition is available, it is checked, whether the body is a final state. A final macro state those regions have terminated, propagates the return code `DEAD` to its own surrounding region. Otherwise the method will return `PAUSE` and the evaluation of the cell is finished.

There is still some work to to, if a transition was enabled.

```
int exit() {
    if (!this.isTransient) {
        this.deepExit();
    }
    this.setNextCell(this.exitHandler.nextCell());
    this.exitHandler.execute();
    return DONE;
}
```

The method `deepExit()` is only called, if the state cell is not transient (transient cells do not execute their onentry actions, either). The exit handler contains the right transition handler. The next cell is set and the transition is executed. `DONE` is returned, so that the surrounding stg can continue with the target.

```
void deepExit() {
    if (!this.body.hasDeepHistory()) {
        this.body.deepExit();
    }
    if (!onExit.isEmpty()) {
        this.onExit.execute();
    }
}
```

The `deepExit()` method is used to recursively execute the onexit actions of all active child states. These actions are only executed, if the body state does not contain a history connector.

5 Future Work

The following list shows possible valuable extensions:

1. Additional primitive types, such as *float*, *double* and *long*.
2. Upgrading the exception handling.
3. Reintegration of parsers for the different kind of labels into the conversion routine in order to avoid conflicts with other modules.
4. The `next` operator to emit a signal in the next instant.
5. *Run modules* to introduce reusability of components and reduce redundancy.
6. An interface to a host language to access external defined functions and types.
7. Detect concurrent variable read/write issues.
8. *Textual macro states* to support inline Esterel code (perhaps by using the diploma thesis of *Lars Kühl* [12] to convert Esterel code into SyncCharts).
9. Refinement of the potential computation. Every refinement will increase the set of syncCharts that can be simulated.
10. Modifications to the simulator to perform a static constructiveness analysis.

6 Conclusions

SyncCharts have a rather simple syntax, but the resulting behavior can be arbitrary complex. Compared to Harel's Statecharts the determinism of SyncCharts is very appealing, despite of the necessity to deal with constructiveness. Experience in synchronous programming with Esterel is a good prerequisite in order to work with SyncCharts.

The different components of SyncCharts are mapped to the design of the syncChart simulator. The semantics of SyncCharts is described and implemented in a comprehensible operational way. The results of the simulation are very detailed and the separate KIEL browser can choose how detailed the results are finally displayed to the user.

Bibliography

[1] Charles André. Representation and Analysis of Reactive Behaviors: A Synchronous Approach. In *Computational Engineering in Systems Applications (CESA)*, pages 19–29, Lille (F), July 1996. IEEE-SMC.

[2] Charles André. SyncCharts: A Visual Representation of Reactive Behaviors. Technical Report RR 95–52, rev. RR (96–56), I3S, Sophia-Antipolis, France, Rev. April 1996.

[3] Charles André. Computing SyncCharts Reactions. In *Electronic Notes in Theoretical Computer Science*, volume 88. Elsevier, July 2003.

[4] Charles André. Semantics of S.S.M (Safe State Machine). Technical report, I3S, Sophia-Antipolis, France, 2003.

[5] Gerard Berry. *The Esterel v5 Language Primer*, 1999.

[6] Gerard Berry. The foundations of Esterel. *Proof, Language and Interaction: Essays in Honour of Robin Milner*, 2000. Editors: G. Plotkin, C. Stirling and M. Tofte.

[7] Gerard Berry and Georges Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[8] Esterel Technologies. Esterel studio. `http://www.esterel-technologies.com/products/esterel-studio/overview.html`.

[9] D. Harel and M. Politi. Modeling Reactive Systems with Statecharts: The Statemate Approach. Technical Report D-1100-43, i-Logix, Three Riverside Drive, Andover, MA 01810, USA, June 1996.

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

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

[12] Lars Kuehl. Transformation von Esterel nach Esterel Studio. Diplomarbeit, Christian-Albrechts-Universität zu Kiel, Institut für Informatik und Praktische Mathematik, September 2005.

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

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

[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.

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

Source Code

.1 kiel.configMngr

.1.1 ChangeValue.java

```java
/*
 * Created on 27.11.2004
 */
package kiel.configMngr;

/**
 * ChangeValue is the basic class for all micro steps concerning value changes.
 * @author Andre Ohlhoff
 */
public abstract class ChangeValue implements MicroStep {

    /**
     * the object which value changed.
     */
    private Object object = null;

    /**
     * the value of the object.
     */
    private Integer value = null;

    /**
     * the value of the object represented as string.
     */
    private String stringvalue = null;

    /**
     * Constructs a ChangeValue micro step.
     * @param o the object
     * @param v the value
     */
    public ChangeValue(final Object o, final Integer v) {
        this.object = o;
        this.value = v;
        if (v != null) {
            this.stringvalue = v.toString();
        } else {
            this.stringvalue = null;
        }
    }

    /**
     * Constructs a ChangeValue micro step.
     * @param o the object
     * @param s the string value
     */
    public ChangeValue(final Object o, final String s) {
        this.object = o;
        this.stringvalue = s;

        if (s != null) {
            try {
                value = new Integer((int) Double.parseDouble(s));
            } catch (NumberFormatException e) {
                value = new Integer(0);
            }
        } else {
            this.value = null;
        }
    }

    /**
     * Returns the object.
     * @return Returns the object.
     */
    public final Object getObject() {
        return this.object;
    }

    /**
     * Sets the object.
     * @param o The object to set.
     */
    public final void setObject(final Object o) {
        this.object = o;
    }

    /**
     * Returns the value (might be null).
     * @return Returns the value.
     */
    public final Integer getValue() {
        return this.value;
    }

    /**
     * Returns the string value (might be null).
     * @return Returns the string value.
     */
    public final String getStringValue() {
        return this.stringvalue;
    }

    /**
     * Sets the value.
     * @param v The value to set.
     */
    public final void setValue(final Integer v) {
        this.value = v;
    }
}
```

.1.2 ConfigMngr.java

```java
package kiel.configMngr;

import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;

import kiel.dataStructure.ANDState;
import kiel.dataStructure.DynamicChoice;
import kiel.dataStructure.InitialState;
import kiel.dataStructure.Node;
import kiel.dataStructure.ORState;
import kiel.dataStructure.Region;
import kiel.dataStructure.SimpleState;
import kiel.dataStructure.StateChart;

/**
 * The ConfigMngr computes all configurations of a kiel statechart.
 * @author Andre Ohlhoff
 */
public class ConfigMngr {

    /**
     * Returns all configurations of the given statechart.
     * @param statechart kiel statechart
     * @return Returns all configurations of the given statechart.
     */
    public final Configuration[] getConfigs(final StateChart statechart) {
        ArrayList temp = this.compute(statechart.getRootNode());
        temp.remove(temp.size() - 1);
        Configuration[] configs = new Configuration[temp.size()];

        for (int i = 0; i < configs.length; i++) {
            configs[i] = new Configuration((ArrayList) temp.get(i));
        }

        return configs;
    }

    /**
     * Returns a list of lists of nodes. The lists of nodes correspond to a
     * possible configuration within the given node.
     * @param node kiel node
     * @return Returns a list of lists of nodes.
     */
    private ArrayList compute(final Node node) {
        if (node instanceof ANDState) {
            return this.computeAND((ANDState) node);
        } else if (node instanceof ORState) {
            return this.computeOR((ORState) node);
        } else if (node instanceof Region) {
            return this.computeRegion(((Region) node).getSubnodes());
        } else if (node instanceof SimpleState) {
            return this.computeSimple(node);
        } else if (node instanceof InitialState) {
            return this.computeSimple(node);
        } else if (node instanceof DynamicChoice) {
            return this.computeSimple(node);
        } else {
            return new ArrayList();
        }
    }

    /**
     * Returns a list of lists of nodes. The lists of nodes correspond to a
     * possible configuration within the given ANDState.
     * @param state kiel ANDState
     * @return Returns a list of lists of nodes. The lists of nodes correspond
     *         to a possible configuration within the given ANDState.
     */
    private ArrayList computeAND(final ANDState state) {
        ArrayList temp = new ArrayList();
        Iterator iter;

        iter = state.getSubnodes().iterator();
        while (iter.hasNext()) {
            Node node = (Node) iter.next();
            temp.add(this.compute(node));
        }
        ArrayList combined = this.combine(temp);

        ArrayList t = new ArrayList();
        t.add(state);
        combined.add(t);

        return combined;
    }

    /**
     * This method takes a list of lists of lists of nodes and combines the
     * lists of lists of nodes to lists of nodes. Here an example in an Haskell
     * like notation: [[[A,B],[A,C]],[[D,E],[F,E]]] is combined to
     * [[A,B,D,E],[A,B,F,E],[A,C,D,E],[A,C,F,E]]. (A,B,C,D,E and F are nodes)
     * @param list List of lists of lists of nodes
     * @return List of lists of nodes.
     */
    private ArrayList combine(final ArrayList list) {
        if (list.size() == 1) {
            return (ArrayList) list.get(0);
        } else {
            ArrayList first = (ArrayList) list.remove(0);
            ArrayList second = (ArrayList) list.remove(0);
            ArrayList combine = this.combine(first, second);
            first = null;
            second = null;
            list.add(0, combine);
            combine = null;
            System.gc();
            return this.combine(list);
        }
    }

    /**
```

```
     * Combines two lists of lists of nodes to one list of lists of nodes. The
     * elements of the first list are pairwise combined with the elements of the
     * second list.
     * @param first first list
     * @param second second list
     * @return list of lists of nodes.
     */
    private ArrayList combine(final ArrayList first, final ArrayList second) {
        ArrayList result = new ArrayList();
        for (int i = 0; i < first.size(); i++) {
            for (int j = 0; j < second.size(); j++) {
                ArrayList config = new ArrayList();
                config.addAll((ArrayList) first.get(i));
                config.addAll((ArrayList) second.get(j));
                result.add(config);
                config = null;
            }
            // System.gc();
            // System.out.println("Free memory : " + runtime.freeMemory());
        }
        return result;
    }

    /**
     * Returns a list of lists of nodes. The lists of nodes correspond to a
     * possible configuration within the given ORState.
     * @param state kiel ORState
     * @return Returns a list of lists of nodes. The lists of nodes correspond
     *         to a possible configuration within the given ORState.
     */
    private ArrayList computeOR(final ORState state) {
        ArrayList result = this.computeRegion(state.getSubnodes());

        ArrayList temp = new ArrayList();
        temp.add(state);
        result.add(temp);

        return result;
    }

    /**
     * Returns a list of lists of nodes. The lists of nodes correspond to a
     * possible configuration within the given nodes of one region.
     * @param nodes nodes of one region
     * @return Returns a list of lists of nodes. The lists of nodes correspond
     *         to a possible configuration within the given nodes of one region.
     */
    private ArrayList computeRegion(final Collection nodes) {
        ArrayList temp = new ArrayList();
        Iterator iter = nodes.iterator();

        while (iter.hasNext()) {
            Node node = (Node) iter.next();
            temp.addAll(this.compute(node));
        }

        return temp;

    }

    /**
     * Returns a list of a list which contains the given node.
     * @param node kiel node (simleState, dynamicChoice or initialState)
     * @return Returns a list of a list which contains the given node.
     */
    private ArrayList computeSimple(final Node node) {
        ArrayList temp1 = new ArrayList();
        ArrayList temp2 = new ArrayList();
        temp2.add(node);
        temp1.add(temp2);
        return temp1;
    }

}
```

.1.3 Configuration.java

```java
/*
 * Created on 07.06.2004
 */
package kiel.configMngr;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Comparator;

import kiel.dataStructure.Node;
import kiel.dataStructure.PseudoState;

/**
 * A Configuration describes the status of a statechart. A Configuration
 * contains the set of states which are active. To simplify matters the
 * Configuration only contains states which don't contain additional active
 * states.
 * @author Andre Ohlhoff
 */
public class Configuration implements java.io.Serializable {

    /**
     * UID.
     */
    private static final long serialVersionUID = 1L;

    /**
     * This flag is true, if this Configuration contains PseudoStates.
     */
    private boolean isPseudo = false;

    /**
     * The 'set' of nodes within this Configuration.
     */
    private Node[] nodes = new Node[0];

    /**
     * Constructs a new Configuration with the given nodes.
     * @param n nodes
     */
    public Configuration(final Node[] n) {
        this.nodes = n;
        this.testIfPseudo();
        this.init();
    }

    /**
     * Constructs a new Configuration with a list of nodes.
     * @param n list of nodes.
     */
    public Configuration(final ArrayList n) {
        this((Node[]) n.toArray(new Node[n.size()]));
    }

    /**
     * Returns the nodes of this configuration.
     * @return Returns the nodes of this configuration.
     */
    public final Node[] getNodes() {
        return this.nodes;
    }

    /**
     * Returns true if the node is in this configuration.
     * @param n Node to be compared
     * @return true if the node is in this configuration
     */
    public final boolean contains(final Node n) {
        for (int i = 0; i < nodes.length; i++) {
            if (nodes[i].equals(n)) {
                return true;
            }
        }
        return false;
    }

    /**
     * Returns a readable string representation.
     * @return Returns a readable string representation.
     */
    public final String toString() {
        StringBuffer toStringBuffer = new StringBuffer();
        if (this.isPseudo) {
            toStringBuffer.append("pseudo:␣");
        }
        for (int i = 0; i < this.nodes.length; i++) {
            Node node = this.nodes[i];
            String temp = node.getName();
            if (temp.equals("")) {
                toStringBuffer.append(node.getID());
            } else {
                toStringBuffer.append(node.getName());
            }
            // }
            if (i + 1 < this.nodes.length) {
                toStringBuffer.append("␣");
            }
        }
        return toStringBuffer.toString();
    }

    /**
     * Sets the isPseudo flag to true, if one of the nodes is a PseudoState.
     */
    private void testIfPseudo() {
        for (int i = 0; i < this.nodes.length; i++) {
            if (nodes[i] instanceof PseudoState) {
                this.isPseudo = true;
                return;
            }
        }
    }
    /**
     * Creates the readable toString and the unique hashString.
     */
    private void init() {
        Comparator comp = new Comparator() {
            public int compare(final Object o1, final Object o2) {
                Node n1 = (Node) o1;
                Node n2 = (Node) o2;
                return n1.getID().compareTo(n2.getID());
            }
        };

        Arrays.sort(this.nodes, comp);

    }

    /**
     * Compares this Configuration to the given object.
     * @param o the reference object with which to compare.
     * @return if this Configuration is the same as the o argument; false
     *         otherwise.
     */
    public final boolean equals(final Object o) {
        if (o == null) {
            return false;
        }
        return this.hashString().equals(((Configuration) o).hashString());
    }

    /**
     * Returns the hashcode of the Configuration.
     * @return Returns the hashcode of the Configuration.
     */
    public final int hashCode() {
        return this.hashString().hashCode();
    }

    /**
     * Return a String used to compute the hashcode.
     * @return Return a String used to compute the hashcode.
     */
    private String hashString() {
        StringBuffer hashStringBuffer = new StringBuffer();
        for (int i = 0; i < this.nodes.length; i++) {
            Node node = this.nodes[i];
            hashStringBuffer.append(node.getID());
        }
        return hashStringBuffer.toString();
    }

}
```

.1.4 ExecuteTransition.java

```java
/*
 * Created on 26.11.2004
 */
package kiel.configMngr;

import kiel.dataStructure.Transition;

/**
 * ExcecuteTransition is generated if a transition is executed.
 * @author Andre Ohlhoff
 */
public class ExecuteTransition extends TransitionStatus {

    /**
     * Constructs a new ExecuteTransition micro step.
     * @param t
     *            transition that is executed.
     */
    public ExecuteTransition(final Transition t) {
        super(t);
    }

    /**
     * Returns a simple message.
     * @return Returns a simple message.
     */
    public final String toString() {
        StringBuffer result = new StringBuffer();
        result.append("executing:␣");
        result.append(this.getTransition().toString());
        return result.toString();
    }

}
```

.1.5 MacroStep.java

```
package kiel.configMngr;

import java.util.LinkedList;

/**
 * A MacroStep contains a list of all MicroSteps of one reaction step or
 * instant.
 * @author Andre Ohlhoff
 */
public class MacroStep {

    /**
     * number of the instant / step.
     */
    private final int instant;

    /**
     * List of MicroSteps.
     */
    private LinkedList microSteps = new LinkedList();

    /**
     * Contructs a new MacroStep with the given instant number.
     * @param i number of the instant.
     */
    public MacroStep(final int i) {
        this.instant = i;
    }

    /**
     * Adds a MicroStep.
     * @param step a MicroStep
     */
    public final void addMicroStep(final MicroStep step) {
        this.microSteps.addLast(step);
    }

    /**
     * Returns the list of MicroSteps.
     * @return Returns the list of MicroSteps.
     */
    public final LinkedList getMicroSteps() {
        return this.microSteps;
    }

    /**
     * Returns the number of the instant.
     * @return Returns the number of the instant.
     */
    public final int getInstant() {
        return this.instant;
    }
}
```

.1.6 MicroStep.java

```java
/*
 * Created on 19.05.2004
 */
package kiel.configMngr;

/**
 * MicroStep is an empty interface for all kinds of micro steps.
 * @author Andre Ohlhoff
 */
public interface MicroStep {
}
```

.1.7 OnEntry.java

```
/*
 * Created on 26.11.2004
 */
package kiel.configMngr;

import kiel.dataStructure.State;

/**
 * OnEntry is generated if a state executes its onentry actions.
 * @author Andre Ohlhoff
 */
public class OnEntry extends StateStatus {

    /**
     * Constructs an OnEntry micro step.
     * @param s
     *            state that executes its onentry actions.
     */
    public OnEntry(final State s) {
        super(s);
    }

    /**
     * Returns a simple message.
     * @return Returns a simple message.
     */
    public final String toString() {
        StringBuffer result = new StringBuffer();
        result.append("OnEntry:␣");
        String test = this.getState().getName();
        if (test.equals("")) {
            result.append(this.getState().getID());
        } else {
            result.append(test);
        }
        return result.toString();
    }

}
```

.1.8 OnExit.java

```
/*
 * Created on 26.11.2004
 */
package kiel.configMngr;

import kiel.dataStructure.State;

/**
 * OnExit is generated if a state executes its onexit actions.
 * @author Andre Ohlhoff
 */
public class OnExit extends StateStatus {

    /**
     * Constructs an OnExit micro step.
     * @param s
     *            state that executes its onexit actions.
     */
    public OnExit(final State s) {
        super(s);
    }

    /**
     * Returns a simple message.
     * @return Returns a simple message.
     */
    public final String toString() {
        StringBuffer result = new StringBuffer();
        result.append("OnExit:␣");
        String test = this.getState().getName();
        if (test.equals("")) {
            result.append(this.getState().getID());
        } else {
            result.append(test);
        }
        return result.toString();
    }
}
```

.1.9 OnInside.java

```
/*
 * Created on 26.11.2004
 */
package kiel.configMngr;

import kiel.dataStructure.State;

/**
 * OnInside is generated if a state executes its oninside actions.
 * @author Andre Ohlhoff
 */
public class OnInside extends StateStatus {

    /**
     * Constructs an OnInside micro step.
     * @param s
     *            state that executes its oninside actions.
     */
    public OnInside(final State s) {
        super(s);
    }

    /**
     * Returns a simple message.
     * @return Returns a simple message.
     */
    public final String toString() {
        StringBuffer result = new StringBuffer();
        result.append("OnInside:␣");
        String test = this.getState().getName();
        if (test.equals("")) {
            result.append(this.getState().getID());
        } else {
            result.append(test);
        }
        return result.toString();
    }

}
```

.1.10 SignalAbsent.java

```
/*
 * Created on 26.11.2004
 */
package kiel.configMngr;

import kiel.dataStructure.eventexp.Signal;

/**
 * The SignalAbsent micro step says that a specific signal is now known to be
 * absent.
 * @author Andre Ohlhoff
 */
public class SignalAbsent extends SignalStatus {

    /**
     * Constructs a SignalAbsent micro step.
     * @param s Signal that is absent
     */
    public SignalAbsent(final Signal s) {
        super(s);
    }

    /**
     * Returns a simple string message.
     * @return Returns a simple string message.
     */
    public final String toString() {
        StringBuffer result = new StringBuffer();
        result.append("signal␣");
        result.append(this.getSignal().getName());
        result.append("␣is␣absent!");
        return result.toString();
    }
}
```

.1.11 SignalPresent.java

```
/*
 * Created on 26.11.2004
 */
package kiel.configMngr;

import kiel.dataStructure.eventexp.Signal;

/**
 * The SignalPresent micro step says that a specific signal is now present.
 * @author Andre Ohlhoff
 */
public class SignalPresent extends SignalStatus {

    /**
     * Constructs a SignalPresent micro step.
     * @param s Signal that is present
     */
    public SignalPresent(final Signal s) {
        super(s);
    }

    /**
     * Returns a simple string message.
     * @return Returns a simple string message.
     */
    public final String toString() {
        StringBuffer result = new StringBuffer();
        result.append("signal␣");
        result.append(this.getSignal().getName());
        result.append("␣is␣present!");
        return result.toString();
    }
}
```

.1.12 SignalStatus.java

```java
/*
 * Created on 26.11.2004
 */
package kiel.configMngr;

import kiel.dataStructure.eventexp.Signal;

/**
 * The SignalStatus micro step is the basic class for all signal status changing
 * micro steps.
 * @author Andre Ohlhoff
 */
public abstract class SignalStatus implements MicroStep {

    /**
     * Signal that is concerned.
     */
    private Signal signal = null;

    /**
     * Constructs a SignalStatus micro step.
     * @param s Signal that is concerned
     */
    public SignalStatus(final Signal s) {
        this.signal = s;
    }

    /**
     * Returns the signal.
     * @return Returns the signal.
     */
    public final Signal getSignal() {
        return this.signal;
    }

    /**
     * Sets the signal.
     * @param s The signal to set.
     */
    public final void setSignal(final Signal s) {
        this.signal = s;
    }
}
```

.1.13 SignalUnknown.java

```
/*
 * Created on 26.11.2004
 */
package kiel.configMngr;

import kiel.dataStructure.eventexp.Signal;

/**
 * The SignalUnknown micro step says that a specific signal is now unknown.
 * @author Andre Ohlhoff
 */
public class SignalUnknown extends SignalStatus {

    /**
     * Constructs a SignalUnknown micro step.
     * @param s Signal that is unknown
     */
    public SignalUnknown(final Signal s) {
        super(s);
    }

    /**
     * Returns a simple string message.
     * @return Returns a simple string message.
     */
    public final String toString() {
        StringBuffer result = new StringBuffer();
        result.append("signal␣");
        result.append(this.getSignal().getName());
        result.append("␣is␣unknown!");
        return result.toString();
    }
}
```

.1.14 SignalValue.java

```java
/*
 * Created on 27.11.2004
 */
package kiel.configMngr;

import kiel.dataStructure.eventexp.Signal;

/**
 * The SignalValue micro step is generated if a signal changes its value or if
 * the value of a signals stays like it was in the last instant.
 * @author Andre Ohlhoff
 */
public class SignalValue extends ChangeValue {

    /**
     * Constructs a SignalValue micro step.
     * @param s the signal
     * @param v the value
     */
    public SignalValue(final Signal s, final Integer v) {
        super(s, v);
    }

    /**
     * Returns a simple message.
     * @return Returns a simple message.
     */
    public final String toString() {
        StringBuffer result = new StringBuffer();
        result.append("value␣of␣signal␣");
        result.append(((Signal) this.getObject()).getName());
        result.append("␣is␣");
        result.append(this.getValue());
        return result.toString();
    }

}
```

.1.15 StateActivated.java

```java
/*
 * Created on 26.11.2004
 */
package kiel.configMngr;

import kiel.dataStructure.State;

/**
 * StateActivated is generated if a state is activated.
 * @author Andre Ohlhoff
 */
public class StateActivated extends StateStatusAndConfig {

    /**
     * Constructs a StateActivated micro step.
     * @param s state that is activated
     * @param c the new configuration
     */
    public StateActivated(final State s, final Configuration c) {
        super(s, c);
    }

    /**
     * Returns a simple message.
     * @return Returns a simple message.
     */
    public final String toString() {
        StringBuffer result = new StringBuffer();
        result.append("activated:␣");
        String test = this.getState().getName();
        if (test.equals("")) {
            result.append(this.getState().getID());
        } else {
            result.append(test);
        }
        result.append(",␣configuration:␣");
        result.append(this.getConfiguration().toString());
        return result.toString();
    }
}
```

.1.16 StateDeactivated.java

```java
/*
 * Created on 26.11.2004
 */
package kiel.configMngr;

import kiel.dataStructure.State;

/**
 * StateActivated is generated if a state is deactivated.
 * @author Andre Ohlhoff
 */
public class StateDeactivated extends StateStatus {

    /**
     * Constructs a StateDeactivated micro step.
     * @param s state that is activated
     */
    public StateDeactivated(final State s) {
        super(s);
    }

    /**
     * Returns a simple message.
     * @return Returns a simple message.
     */
    public final String toString() {
        StringBuffer result = new StringBuffer();
        result.append("deactivated:␣");
        String test = this.getState().getName();
        if (test.equals("")) {
            result.append(this.getState().getID());
        } else {
            result.append(test);
        }
        return result.toString();
    }

}
```

.1.17 StateNotSuspended.java

```
/*
 * Created on 26.11.2004
 */
package kiel.configMngr;

import kiel.dataStructure.State;

/**
 * StateSuspended is generated if a state is not suspended in this instant, but
 * was in the last instant.
 * @author Andre Ohlhoff
 */
public class StateNotSuspended extends StateStatusAndConfig {

    /**
     * Constructs a new StateNotSuspended.
     * @param s state that is suspended.
     * @param c the new configuration.
     */
    public StateNotSuspended(final State s, final Configuration c) {
        super(s, c);
    }

    /**
     * Returns a simple message.
     * @return Returns a simple message.
     */
    public final String toString() {
        StringBuffer result = new StringBuffer();
        result.append("Not␣Suspended:␣");
        String test = this.getState().getName();
        if (test.equals("")) {
            result.append(this.getState().getID());
        } else {
            result.append(test);
        }
        result.append(",␣configuration:␣");
        result.append(this.getConfiguration().toString());
        return result.toString();

    }
}
```

.1.18 StateStatus.java

```
/*
 * Created on 26.11.2004
 */
package kiel.configMngr;

import kiel.dataStructure.State;

/**
 * The StateStatus micro step is the basic class for all micro steps concerning
 * states.
 * @author Andre Ohlhoff
 */
public abstract class StateStatus implements MicroStep {

    /**
     * The state that is concerned.
     */
    private State state = null;

    /**
     * Constructs a StateStatus micro step.
     * @param s state that is concerned
     */
    public StateStatus(final State s) {
        this.state = s;
    }

    /**
     * Returns the state.
     * @return Returns the state.
     */
    public final State getState() {
        return this.state;
    }

    /**
     * Sets the state.
     * @param s The state to set.
     */
    public final void setState(final State s) {
        this.state = s;
    }
}
```

.1.19 StateStatusAndConfig.java

```
/*
 * Created on 17.12.2004
 */
package kiel.configMngr;

import kiel.dataStructure.State;

/**
 * The StateStatus micro step is the basic class for all micro steps concerning
 * states and configurations.
 * @author Andre Ohlhoff
 */
public abstract class StateStatusAndConfig extends StateStatus {

    /**
     * The new configuration.
     */
    private Configuration config = null;

    /**
     * Constructs a StateStatusAndConfig micro step.
     * @param s the state
     * @param c the configuration
     */
    public StateStatusAndConfig(final State s, final Configuration c) {
        super(s);
        this.config = c;
    }

    /**
     * Returns the configuration.
     * @return Returns the configuration.
     */
    public final Configuration getConfiguration() {
        return this.config;
    }

}
```

.1.20 StateSuspended.java

```java
/*
 * Created on 26.11.2004
 */
package kiel.configMngr;

import kiel.dataStructure.State;

/**
 * StateSuspended is generated if a state is suspended.
 * @author Andre Ohlhoff
 */
public class StateSuspended extends StateStatusAndConfig {

    /**
     * Constructs a new StateSuspended.
     * @param s state that is suspended.
     * @param c the new configuration.
     */
    public StateSuspended(final State s, final Configuration c) {
        super(s, c);
    }

    /**
     * Returns a simple message.
     * @return Returns a simple message.
     */
    public final String toString() {
        StringBuffer result = new StringBuffer();
        result.append("Suspended:␣");
        String test = this.getState().getName();
        if (test.equals("")) {
            result.append(this.getState().getID());
        } else {
            result.append(test);
        }
        result.append(",␣configuration:␣");
        result.append(this.getConfiguration().toString());
        return result.toString();
    }
}
```

.1.21 StateTransient.java

```
/*
 * Created on 26.11.2004
 */
package kiel.configMngr;

import kiel.dataStructure.State;

/**
 * StateTransient is generated if a state is transient.
 * @author Andre Ohlhoff
 */
public class StateTransient extends StateStatus {

    /**
     * Constructs a new StateTransient.
     * @param s state that is transient.
     */
    public StateTransient(final State s) {
        super(s);
    }

    /**
     * Returns a simple message.
     * @return Returns a simple message.
     */
    public final String toString() {
        StringBuffer result = new StringBuffer();
        result.append("Transient:␣");
        String test = this.getState().getName();
        if (test.equals("")) {
            result.append(this.getState().getID());
        } else {
            result.append(test);
        }
        return result.toString();
    }

}
```

.1.22 TestTransition.java

```java
/*
 * Created on 26.11.2004
 */
package kiel.configMngr;

import kiel.dataStructure.Transition;

/**
 * TestTransition is generated if a transition is tested.
 * @author Andre Ohlhoff
 */
public class TestTransition extends TransitionStatus {

    /**
     * Constructs a TestTransition micro step.
     * @param t transition that is tested.
     */
    public TestTransition(final Transition t) {
        super(t);
    }

    /**
     * Returns a simple message.
     * @return Returns a simple message.
     */
    public final String toString() {
        StringBuffer result = new StringBuffer();
        result.append("testing:␣");
        result.append(this.getTransition().toString());
        return result.toString();
    }

}
```

.1.23 TransitionStatus.java

```java
/*
 * Created on 26.11.2004
 */
package kiel.configMngr;

import kiel.dataStructure.Transition;

/**
 * TransitionStatus is the basic class for all micro steps concerning
 * transitions.
 * @author Andre Ohlhoff
 */
public abstract class TransitionStatus implements MicroStep {

    /**
     * Transition.
     */
    private Transition transition = null;

    /**
     * Constructs a TransitionStatus micro step.
     * @param t Transition that is concerned
     */
    public TransitionStatus(final Transition t) {
        this.transition = t;
    }

    /**
     * Returns the transition.
     * @return Returns the transition.
     */
    public final Transition getTransition() {
        return this.transition;
    }

    /**
     * Sets the transition.
     * @param t The transition to set.
     */
    public final void setTransition(final Transition t) {
        this.transition = t;
    }
}
```

.1.24 VariableValue.java

```java
/*
 * Created on 27.11.2004
 */
package kiel.configMngr;

import kiel.dataStructure.Variable;

/**
 * A VariableValue micro step is generated if a variable changes its value.
 * @author Andre Ohlhoff
 */
public class VariableValue extends ChangeValue {

    /**
     * Constructs a VariableChanged micro step.
     * @param var the variable
     * @param v the value
     */
    public VariableValue(final Variable var, final Integer v) {
        super(var, v);
    }

    /**
     * Constructs a VariableChanged micro step for string variables.
     * @param var the variable
     * @param v the string value
     */
    public VariableValue(final Variable var, final String v) {
        super(var, v);
    }

    /**
     * Returns a simple message.
     * @return Returns a simple message.
     */
    public final String toString() {
        StringBuffer result = new StringBuffer();
        result.append("value␣of␣variable␣");
        result.append(((Variable) this.getObject()).getName());
        result.append("␣is␣");
        result.append(this.getValue());
        return result.toString();
    }
}
```

.2 kiel.simulator

.2.1 Simulator.java

```
/*
 * Created on 14.05.2004
 */
package kiel.simulator;

import java.util.Collection;

import kiel.configMngr.MacroStep;
import kiel.dataStructure.StateChart;
import kiel.dataStructure.eventexp.Event;
import kiel.dataStructure.Variable;
import kiel.util.LogFile;

/**
 * Interface for different simulators.
 * @author Andre Ohlhoff
 */
public interface Simulator {

    /**
     * Sets the logFile for simulator output.
     * @param log logFile for simulator output.
     */
    void setLogFile(LogFile log);

    /**
     * Resets the simulator.
     */
    void reset();

    /**
     * Computes the next step. Before calling this method, use the two emit
     * methods to set the statuses of the input signals for this step. Notice
     * that there ist no initialstep or initialization necessary. The simulation
     * starts with instance 0.
     * @return Returns a MacroStep as a summary of this step.
     * @throws SimulatorException An Exception is thrown if this step cannot be
     *             computed.
     */
    MacroStep nextStep() throws SimulatorException;

    /**
     * Set the statechart to simulate.
     * @param sc The statechart to simulate.
     * @throws SimulatorException An Exception is thrown if the statechart is
     *             obviously wrong.
     */
    void setStateChart(final StateChart sc) throws SimulatorException;

    /**
     * Returns a collection of the input events.
     * @return Returns a collection of the input events.
     */
    Collection getInput();

    /**
     * Returns a collection of the output events.
     * @return Returns a collection of the output events.
     */
    Collection getOutput();

    /**
     * Sets the status of the given event/signal to present for the next
     * step/instant.
     * @param e Event to set present for next instant.
     * @throws SimulatorException An Exception is thrown if the event cannot be
     *             emitted.
     */
    void emit(final Event e) throws SimulatorException;

    /**
     * Sets the status of the given event/signal to present with the given value
     * for the next step/instant.
     * @param e Event to set present for next instant.
     * @param i Value for the event.
     * @throws SimulatorException An Exception is thrown if the event cannot be
     *             emitted.
     */
    void emit(final Event e, final int i) throws SimulatorException;

    /**
     * Sets the value of the given variable to the given value. The value is
     * valid until it is set again.
     * @param var = Variable for which the new value has to be set.
     * @param s = The new value for the varibale represented as a String.
     * @throws SimulatorException An error occured.
     */
    void setVariable(final Variable var, final String s)
            throws SimulatorException;

    /**
     * Sets the status of the given feature.
     * @param feature The feature name.
     * @param status The status of the feature.
     * @throws SimulatorException When a feature is not supported.
     */
    void setFeature(final String feature, final boolean status)
            throws SimulatorException;

    /**
     * Returns the status of the given feature.
     * @param feature The feature name.
     * @return Returns the status of the given feature.
     * @throws SimulatorException When a feature is not supported.
     */
    boolean getFeature(final String feature) throws SimulatorException;
}
```

.2.2 SimulatorChoser.java

```
/*
 * Created on 01.11.2004
 */
package kiel.simulator;

import kiel.simulator.syncchart.STRLSimulator;
import kiel.simulator.matlab.StateflowSimulator;

/**
 * The SimulatorChoser creates adequate simulators.
 * @author Andre Ohlhoff
 */
public abstract class SimulatorChoser {

    /**
     * Used by the stateflowsimulator.
     */
    public static final String SIMULATES_STRING_LABEL
            = "StringLabel_Simulation";

    /**
     * Used by the stateflowsimulator.
     */
    public static final String INPUT_EVENTS_SUPPORT_LABEL
            = "InputEvents_Simulation";

    /**
     * Useless.
     */
    private SimulatorChoser() {
    }

    /**
     * Returns an simulator matching to the given modelSource and modelVersion.
     * If no matching simulator is found, null is returned.
     * @param modelSource source
     * @param modelVersion version
     * @return Returns an simulator matching to the given modelSource and
     *         modelVersion.
     */
    public static Simulator getSimulator(final String modelSource,
            final String modelVersion) {
        if (modelSource.equalsIgnoreCase("Esterel␣Studio")
                && modelVersion.equalsIgnoreCase("5.0")) {
            return new STRLSimulator();
        } else if (modelSource.equalsIgnoreCase("Matlab/Stateflow")
                && modelVersion.equalsIgnoreCase("6.1")) {
            return new StateflowSimulator();
        } else {
            return null;
        }
    }

}
```

.2.3 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 {

    /**
     *
     */
    private static final long serialVersionUID = 1L;

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

}
```

.3 kiel.simulator.syncchart

.3.1 STRLComponent.java

```java
/*
 * Created on 15.10.2004
 */
package kiel.simulator.syncchart;

import java.util.Collection;
import java.util.Iterator;

/**
 * STRLComponent is the basic class for all components which need a reference to
 * the simulator.
 * @author Andre Ohlhoff
 */
public abstract class STRLComponent {

    /**
     * Sets sim as context for all components.
     * @param sim
     *            STRLSimulator
     * @param components
     *            components which need a reference to the simulator.
     */
    public static void setContext(final STRLSimulator sim,
            final STRLComponent[] components) {
        for (int i = 0; i < components.length; i++) {
            components[i].setContext(sim);
        }
    }

    /**
     * Sets sim as context for all components.
     * @param sim
     *            STRLSimulator
     * @param components
     *            components which need a reference to the simulator.
     */
    public static void setContext(final STRLSimulator sim,
            final Collection components) {
        Iterator iter = components.iterator();
        while (iter.hasNext()) {
            ((STRLComponent) iter.next()).setContext(sim);
        }
    }

    /**
     * Reference to the main simulator.
     */
    private STRLSimulator context = null;

    /**
     * Returns the simulator.
     * @return Returns the simulator.
     */
    public final STRLSimulator getContext() {
        return this.context;
    }

    /**
     * Sets the context for the component and all embedded components.
     * @param sim
     *            STRLSimulator
     */
    public final void setContext(final STRLSimulator sim) {
        this.context = sim;
        this.setComponentsContext(sim);
    }

    /**
     * This method is overridden by all components which contain additional
     * componets.
     * @param sim
     *            STRLSimulator.
     */
    public void setComponentsContext(final STRLSimulator sim) {
    }
}
```

.3.2 STRLConditionalCell.java

```java
/*
 * Created on 15.09.2004
 */
package kiel.simulator.syncchart;

import kiel.dataStructure.DynamicChoice;

/**
 * A STRLConditionalCell implements the behavior of the conditional connector.
 * @author Andre Ohlhoff
 */
public class STRLConditionalCell extends STRLPseudoCell {

    /**
     * Constructs a STRLConditionalCell.
     * @param state
     *             kiel DynamicChoice
     */
    public STRLConditionalCell(final DynamicChoice state) {
        super(state);
    }

}
```

.3.3 STRLInitialCell.java

```java
/*
 * Created on 15.09.2004
 */
package kiel.simulator.syncchart;

import kiel.dataStructure.InitialState;

/**
 * A STRLInitialCell implements the behavior of the intial connector.
 * @author Andre Ohlhoff
 */
public class STRLInitialCell extends STRLPseudoCell {

    /**
     * Constructs a STRLInitialCell.
     * @param state
     *            Kiel InitialState
     */
    public STRLInitialCell(final InitialState state) {
        super(state);
    }

}
```

.3.4 STRLMacroState.java

```java
/*
 * Created on 14.05.2004
 */
package kiel.simulator.syncchart;

import java.util.ArrayList;
import java.util.Collection;
import java.util.HashSet;

import kiel.dataStructure.State;
import kiel.simulator.syncchart.exceptions.STRLException;
import kiel.simulator.syncchart.sigexp.STRLSignal;
import kiel.simulator.syncchart.sigexp.STRLSignalHandler;

/**
 * STRLMacroState implements the behavior of a MacroState. A MacroState contains
 * several STRLSTGs and an optional set of local signals.
 * @author Andre Ohlhoff
 */
public class STRLMacroState extends STRLState {

    /**
     * This constant is used as position marker, showing that the MacroState
     * must initialize its local signals.
     */
    private static final int INITSIGNALS = 0;

    /**
     * This constant is used as position marker, showing that the MacroState
     * must reset its local signals.
     */
    private static final int RESETSIGNALSTATE = 1;

    /**
     * This constant is used as position marker, showing that the MacroState is
     * computing the normal reaction.
     */
    private static final int REACTING = 2;

    /**
     * is true if the state has executed his oninside actions.
     */
    private boolean onInsideExecuted = false;

    /**
     * Stores the last return code of its STRLSTGs to avoid redundant
     * computations.
     */
    private int lastResultOfSTGs = STRLReactingComponent.UNKNOWN;

    /**
     * Shows, what the MacroState is doing now or next.
     */
    private int position = STRLMacroState.INITSIGNALS;

    /**
     * a signalHandler holds the local signals.
     */
    private STRLSignalHandler local = new STRLSignalHandler();

    /**
     * a stgHandler holds the stgs.
     */
    private STRLSTGHandler stgs = new STRLSTGHandler();

    /**
     * Constructs a new STRLMacroState with a reference to the corresponding
     * kiel state.
     * @param state
     *            kiel State
     */
    public STRLMacroState(final State state) {
        super(state);
    }

    /**
     * Sets the stgs.
     * @param s
     *            STRLSTGs to be set
     */
    public final void setSTGs(final STRLSTG[] s) {
        this.stgs.setSTGs(s);
    }

    /**
     * Sets the local signals.
     * @param signals
     *            STRLSignals to be set.
     */
    public final void setLocalSignals(final STRLSignal[] signals) {
        this.local.setSignals(signals);
    }

    /**
     * Sets the context for the components.
     * @param c
     *            STRLSimulator used as context.
     */
    public final void setComponentsContext(final STRLSimulator c) {
        this.local.setContext(c);
        this.stgs.setContext(c);
        this.getOnInside().setContext(c);
    }

    /**
     * Computes the reaction.
     * @return DEAD, PAUSE or UNKNOWN
     * @throws STRLException
     *             The exception is thrown if something went wrong.
     */
    public final int react() throws STRLException {
        if (!this.onInsideExecuted) {
            if (!this.getOnInside().isEmpty()) {
                this.getContext().logOnInside(this.getKielState());
                this.getOnInside().execute();
            }
            this.onInsideExecuted = true;
        }
        switch (this.position) {
        case (STRLMacroState.INITSIGNALS):
            return this.initSignals();
        case (STRLMacroState.RESETSIGNALSTATE):
            return this.resetSignalStatus();
        default:
            return this.reacting();
        }

    }

    /**
     * This method is called if a superordinated cell is exited.
     * @throws STRLException
     *             The exception is thrown if something went wrong.
     */
    public final void deepExit() throws STRLException {
        this.stgs.deepExit();
    }

    /**
     * Returns a list of active States below this macrostate.
     * @return Returns a list of active States below this macrostate.
     */
    public final ArrayList getActiveStates() {
        ArrayList result = new ArrayList();
        if (this.isActive()) {
            result.addAll(this.stgs.getActiveStates());
        } else {
            result.add(this.getKielState());
        }
        return result;
    }

    /**
     * Returns the potential of the macrostate.
     * @param fresh
     *             fresh signals
     * @param unknown
     *             unknown signals
     * @return Returns the potential of the macrostate.
     */
    public final Collection getRemainingPotential(final Collection fresh,
            final Collection unknown) {
        Collection result = new HashSet();

        if (!this.onInsideExecuted) {
            result.addAll(this.getOnInside().getPotential());
        }

        if (this.position <= STRLMacroState.RESETSIGNALSTATE) {
            HashSet fresh2 = new HashSet(fresh);
            HashSet unknown2 = new HashSet(unknown);
            if (this.position == STRLMacroState.RESETSIGNALSTATE) {
                unknown2.addAll(this.local.getSignals());
            } else {
                fresh2.addAll(this.local.getSignals());
            }

            result.addAll(this.stgs.getRemainingPotential(fresh2, unknown2));
        } else {
            result.addAll(this.stgs.getRemainingPotential(fresh, unknown));
        }

        return result;

    }

    /**
     * Returns the complete potential of the macrostate.
     * @param fresh
     *             fresh signals
     * @param unknown
     *             unknown signals
     * @return Returns the complete potential of the macrostate.
     */
    public final Collection getCompletePotential(final Collection fresh,
            final Collection unknown) {

        HashSet result = new HashSet();
        HashSet fresh2 = new HashSet(fresh);
        HashSet unknown2 = new HashSet(unknown);

        if (this.hasDeepHistory() && this.wasActiveBefore()) {
            unknown2.addAll(this.local.getSignals());
        } else {
            fresh2.addAll(this.local.getSignals());
        }

        result.addAll(this.getOnInside().getPotential());
        result.addAll(this.stgs.getCompletePotential(fresh2, unknown2));

        return result;
    }

    /**
     * Initializes the local signals and starts the normal reaction thereafter.
     * @return DEAD, PAUSE or UNKNOWN
     * @throws STRLException
     *             The exception is thrown if something went wrong.
     */
    private int initSignals() throws STRLException {
        this.local.initialize();
        this.position = STRLMacroState.REACTING;
        return this.reacting();

    }

    /**
     * Resets the status of the local signals and starts the normal reaction
     * thereafter.
     * @return DEAD, PAUSE or UNKNOWN
     * @throws STRLException
     *              The exception is thrown if something went wrong.
     */
    private int resetSignalStatus() throws STRLException {
        this.local.resetStatus();
        this.position = STRLMacroState.REACTING;
        return this.reacting();
    }

    /**
     * Computes the normal reaction.
     * @return DEAD, PAUSE or UNKNOWN
     * @throws STRLException
     *              The exception is thrown if something went wrong.
     */
    private int reacting() throws STRLException {
        if (this.lastResultOfSTGs == STRLReactingComponent.UNKNOWN) {
            this.lastResultOfSTGs = this.stgs.react();
        }
        return this.lastResultOfSTGs;
    }

    /**
     * Prepares the macroState for the next instant, if the macrostate is active
     * at the end if this instant.
     */
    public final void prepareForNext() {
        if (this.isActive()) {
            this.onInsideExecuted = false;
            this.lastResultOfSTGs = STRLReactingComponent.UNKNOWN;
            this.position = STRLMacroState.RESETSIGNALSTATE;
            this.stgs.prepareForNext();
        }
    }

    /**
     * This method is called if a superordinated cell is entered. The MacroState
     * is prepared for a reaction.
     * @param thc
     *             is true, it the superordinated cell is entered through a
     *             deephistory connector.
     */
    public final void deepEnter(final boolean thc) {

        this.lastResultOfSTGs = STRLReactingComponent.UNKNOWN;
        this.onInsideExecuted = false;

        if (thc && this.wasActiveBefore()) {
            this.position = STRLMacroState.RESETSIGNALSTATE;
        } else {
            this.position = STRLMacroState.INITSIGNALS;
        }

        this.stgs.deepEnter(thc);
    }

    /**
     * Updates the local signals below this macrostate.
     * @param potential
     *             Set of potentially emitted signals.
     */
    public final void update(final Collection potential) {
        this.local.update(potential);
        this.stgs.update(potential);
    }

    /**
     * Called at the end of an instant to update also local signals in inactive
     * states.
     */
    public final void finalUpdate() {
        this.local.update(new HashSet());
        this.stgs.finalUpdate();
    }

    /**
     * Returns true, if all stgs can reach a final state during computation of
     * the potential.
     * @return Returns true, if all stgs can reach a final state during
     *         computation of the potential.
     */
    public final boolean reachesFinal() {
        return this.stgs.reachesFinal();
    }

    /**
     * Returns the potential of all exit actions below this macrostate, which
     * might execute if a superordinated cell is entered.
     * @return Returns the exit potential.
     */
    public final Collection getExitPotential() {
        if (this.hasDeepHistory()) {
            return new HashSet();
        } else {
            return this.stgs.getExitPotential();
        }
    }

    /**
     * Resets some helping flags.
     */
    public final void resetHelpingFlags() {
        this.stgs.resetHelpingFlags();
    }

}
```

.3.5 STRLPseudoCell.java

```
/*
 * Created on 13.06.2004
 */
package kiel.simulator.syncchart;

import java.util.ArrayList;
import java.util.Collection;
import java.util.HashSet;

import kiel.dataStructure.PseudoState;
import kiel.simulator.syncchart.exceptions.STRLException;
import kiel.simulator.syncchart.transition.STRLImmediateTrans;
import kiel.simulator.syncchart.transition.STRLTransitionHandler;

/**
 * STRLPseudoCell is the basic class for STRLConditionalCell and
 * STRLInitialCell.
 * @author Andre Ohlhoff
 */
public abstract class STRLPseudoCell extends STRLReactiveCell {

    /**
     * this constant is used as position marker, showing that the cell must
     * initiliaze its transitions.
     */
    private static final int INITIALIZING = 0;

    /**
     * this constant is used as position marker, showing that the cell is
     * testing its transitions.
     */
    private static final int CHECKING = 1;

    /**
     * shows, what the cell is doing now or next.
     */
    private int position = STRLPseudoCell.INITIALIZING;

    /**
     * a transition handler holds the outgoing transitions.
     */
    private STRLTransitionHandler handler = new STRLTransitionHandler();

    /**
     * Constructs a STRLPseudoCell with a reference to the corresponding kiel
     * PseudoState.
     * @param state
     *            Kiel PseudoState
     */
    public STRLPseudoCell(final PseudoState state) {
        super(state);
    }

    /**
     * Sets the outgoing transitions.
     * @param t
     *            the outgoing transitions.
     */
    public final void setTrans(final STRLImmediateTrans[] t) {
        this.handler.setTransitions(t);
    }

    /**
     * Sets the context for the transitions.
     * @param c
     *            STRLSimulator used as context.
     */
    public final void setComponentsContext(final STRLSimulator c) {
        this.handler.setContext(c);
    }

    /**
     * Returns the remaining potential.
     * @param fresh
     *            fresh signals
     * @param unknown
     *            unknown signals
     * @return Returns the remaining potential.
     */
    public final Collection getRemainingPotential(final Collection fresh,
            final Collection unknown) {
        return this.handler.getPotential(false, fresh, unknown);
    }

    /**
     * Returns the complete potential.
     * @param fresh
     *            fresh signals
     * @param unknown
     *            unknown signals
     * @return Returns the complete potential.
     */
    public final Collection getCompletePotential(final Collection fresh,
            final Collection unknown) {
        return this.handler.getPotential(false, fresh, unknown);
    }

    /**
     * Starts the reaction.
     * @return Returns UNKNOWN or DONE
     * @throws STRLException
     *             The exception is thrown if something went wrong.
     */
    public final int react() throws STRLException {
        switch (this.position) {
        case (STRLPseudoCell.INITIALIZING):
            return this.initialize();
        case (STRLPseudoCell.CHECKING):
            return this.checking();
        default: // dead code
            throw new STRLException("oh␣oh");
        }
```

```java
}

/**
 * Initializes the outgoing transitions and starts the normal reaction
 * thereafter.
 * @return Returns UNKNOWN or DONE
 * @throws STRLException
 *             Wird geworfen, falls bei der Simulation ein Fehler auftritt.
 */
private int initialize() throws STRLException {
    this.handler.init();
    this.position = STRLPseudoCell.CHECKING;
    return this.checking();

}

/**
 * Tests the outgoing transitions.
 * @return Returns UNKNOWN or DONE
 * @throws STRLException
 *             The exception is thrown if something went wrong.
 */
private int checking() throws STRLException {
    int code = this.handler.evaluate(false);
    if (code == STRLTransitionHandler.FALSE) {
        throw new STRLException("pseudostate␣without␣enabled␣transition");
    } else if (code == STRLTransitionHandler.UNKNOWN) {
        return STRLReactingComponent.UNKNOWN;
    } else {
        this.setNextCell(this.handler.nextCell());
        this.handler.execute();
        return STRLReactingComponent.DONE;
    }

}

/**
 * Nothing happens.
 */
public final void deepExit() {
}

/**
 * Nothing happens.
 */
public void prepareForNext() {
}

/**
 * Is called, if the cell will be entered.
 * @param thc
 *            is ignored
 */
public final void enter(final boolean thc) {
    this.position = STRLPseudoCell.INITIALIZING;
}

/**
 * Is called, if a superordinated cell will be entered
 * @param thc
 *            is ignored
 */
public final void deepEnter(final boolean thc) {
    this.enter(thc);
}

/**
 * Returns false.
 * @return Returns false.
 */
public final boolean hasDeepHistory() {
    return false;
}

/**
 * Returns a list containing the corresponding kiel PseudoState.
 * @return Returns a list containing the corresponding kiel PseudoState.
 */
public final ArrayList getActiveStates() {
    ArrayList result = new ArrayList();
    result.add(this.getKielNode());
    return result;
}

/**
 * nothing happens.
 * @param potential
 *            is ignored
 */
public final void update(final Collection potential) {
}

/**
 * nothing happens.
 */
public final void finalUpdate() {

}

/**
 * Returns false.
 * @return Returns false.
 */
public final boolean reachesFinal() {
    return false;
}

/**
 * Returns an empty set.
 * @return Returns an empty set.
 */
public final Collection getExitPotential() {
    return new HashSet();
}

/**
 * Returns an empty set.
 * @return Returns an empty set.
 */
public final Collection getCompleteExitPotential() {
        return new HashSet();
    }

    /**
     * Nothing happens.
     */
    public final void resetHelpingFlags() {
    }
}
```

.3.6 STRLReactingComponent.java

```
/*
 * Created on 15.10.2004
 */
package kiel.simulator.syncchart;

import java.util.ArrayList;
import java.util.Collection;

import kiel.simulator.syncchart.exceptions.STRLException;

/**
 * ReactingComponent contains the the basic return codes.
 * @author Andre Ohlhoff
 */
public abstract class STRLReactingComponent extends STRLComponent {

    /**
     * DONE is returned by a cell, if the cell has enabled a transition.
     */
    public static final int DONE = 0;

    /**
     * DEAD is returned by a SimpleState, which is final. DEAD is also returned
     * by a cell, if the body of the cell returns DEAD and is final. A STG
     * returns DEAD if its current cell returns DEAD. A MacroState returns DEAD,
     * if all contained STGs returned DEAD.
     */
    public static final int DEAD = 1;

    /**
     * A SimpleState returns PAUSE, if it is not final. A cell returns PAUSE, if
     * no transition is enabled. A STG returns PAUSE if the current cell returns
     * PAUSE, and a MacroState returns PAUSE if at least one contained STG
     * returns PAUSE.
     */
    public static final int PAUSE = 2;

    /**
     * A cell returns UNKNOWN if its body returns UNKNOWN or if the value of
     * some trigger is UNKNOWN. A STG returns UNKNOWN if its current cell
     * returns UNKNOWN and a MacroState returns UNKNOWN if at least one of its
     * contained STGs returns UNKNOWN.
     */
    public static final int UNKNOWN = 3;

    /**
     * Returns the list of active states which is needed to construct a
     * configuration. Hence, the list doesn't contain active states which
     * contain other active states.
     * @return Returns the list of active states below this component.
     */
    public abstract ArrayList getActiveStates();

    /**
     * Prepares the component for the next instant. This method is only used for
     * components, which were active at the end of the previous instant.
     */
    public abstract void prepareForNext();

    /**
     * This method is needed to tell a component, that a superordinated cell is
     * entered.
     * @param thc
     *            through-history-connector is true, if a history connector is
     *            used.
     */
    public abstract void deepEnter(final boolean thc);

    /**
     * This method is used to tell a component, that a superordinated cell is
     * exited.
     * @throws STRLException
     *             An exception is thrown if something went wrong.
     */
    public abstract void deepExit() throws STRLException;

    /**
     * Starts the reaction of the component.
     * @return Returns DONE, DEAD, PAUSE or UNKNOWN.
     * @throws STRLException
     *             An exception is thrown if something went wrong.
     */
    public abstract int react() throws STRLException;

    /**
     * Returns the set of signals, which are potentially emitted by this
     * component. This method is used, if the component is currently active.
     * @param fresh
     *            fresh signals. If a signal is contained in this collection,
     *            its (previous) status is assumed to be UNKNOWN.
     * @param unknown
     *            unknown signals. If a signal is contained in this collection
     *            its current status is assumed to be UNKNOWN, and its previous
     *            is assumed to be the real current status.
     * @return Returns the set of signals, which are potentially emitted by this
     *         component.
     */
    public abstract Collection getRemainingPotential(final Collection fresh,
            final Collection unknown);

    /**
     * Returns the set of signals, which are potentially emitted by this
     * component. This method is used, if the component might become active
     * (again) in this instant.
     * @param fresh
     *            fresh signals. If a signal is contained in this collection,
     *            its (previous) status is assumed to be UNKNOWN.
     * @param unknown
     *            unknown signals. If a signal is contained in this collection
     *            its current status is assumed to be UNKNOWN, and its previous
     *            is assumed to be the real current status.
     * @return Returns the set of signals, which are potentially emitted by this
     *          component.
     */
    public abstract Collection getCompletePotential(final Collection fresh,
            final Collection unknown);

    /**
     * Returns the set of signals, which are emitted (via exit actions) if this
     * component is exited.
     * @return Returns the set of signals, which are emitted (via exit actions)
     *         if this component is exited.
     */
    public abstract Collection getExitPotential();

    /**
     * Updates the status (and some inner flags) of all signals, which are not
     * contained in the given collection.
     * @param potential
     *            Set of potentially emitted signals.
     */
    public abstract void update(final Collection potential);

    /**
     * Called at the end of an instant to update also local signals in inactive
     * states.
     */
    public abstract void finalUpdate();

    /**
     * Returns true, if during the computation of the potential a SimpleState is
     * reached, which is final.
     * @return Returns true, if during the computation of the potential a
     *         SimpleState is reached, which is final.
     */
    public abstract boolean reachesFinal();

    /**
     * Resets some flags for the next instant.
     */
    public abstract void resetHelpingFlags();
}
```

.3.7 STRLReactiveCell.java

```
/*
 * Created on 14.05.2004
 */
package kiel.simulator.syncchart;

import kiel.dataStructure.Node;

/**
 * STRLReactiveCell is the basic class for STRLStateCell and STRLPseudoCell.
 * @author Andre Ohlhoff
 */
public abstract class STRLReactiveCell extends STRLReactingComponent {

    /**
     * nextCell contains the target cell of an enabled outgoing transition.
     */
    private STRLReactiveCell nextCell = null;

    /**
     * corresponding kiel node.
     */
    private Node kielNode = null;

    /**
     * Constructs a new STRLReactiveCell for the given kiel node.
     * @param node kiel Node
     */
    public STRLReactiveCell(final Node node) {
        this.kielNode = node;
    }

    /**
     * Sets the next cell.
     * @param next the next cell.
     */
    public final void setNextCell(final STRLReactiveCell next) {
        this.nextCell = next;
    }

    /**
     * Returns the next cell.
     * @return Returns the next cell.
     */
    public final STRLReactiveCell nextCell() {
        return this.nextCell;
    }

    /**
     * Returns the corresponding kiel node.
     * @return Returns the corresponding kiel node.
     */
    public final Node getKielNode() {
        return this.kielNode;
    }

    /**
     * This method must be called if the cell is about to be entered.
     * @param thc is true, if a history connector is used.
     */
    public abstract void enter(final boolean thc);

    /**
     * Returns true, if the cell has a deephistory connector.
     * @return Returns true, if the cell has a deephistory connector.
     */
    public abstract boolean hasDeepHistory();

}
```

.3.8 STRLSimpleState.java

```
/*
 * Created on 14.05.2004
 */
package kiel.simulator.syncchart;

import java.util.ArrayList;
import java.util.Collection;
import java.util.HashSet;

import kiel.dataStructure.SimpleState;
import kiel.simulator.syncchart.exceptions.STRLException;

/**
 * STRLSimpleState implements the behavior of a simple state.
 * @author Andre Ohlhoff
 */
public class STRLSimpleState extends STRLState {

    /**
     * is true, if the state is reached during computation of the potential
     */
    private boolean isReachedFinal = false;

    /**
     * is true, if the oninside actions have been executed this instant.
     */
    private boolean onInsideExecuted = false;

    /**
     * Constructs a STRLSimpleState with a reference to the corresponding kiel
     * SimpleState.
     * @param state
     *            Kiel SimpleState
     */
    public STRLSimpleState(final SimpleState state) {
        super(state);
    }

    /**
     * Starts the reaction.
     * @return DEAD or PAUSE
     * @throws STRLException
     *             The exception is thrown if something went wrong.
     */
    public final int react() throws STRLException {
        if (this.isFinal()) {
            return STRLReactingComponent.DEAD;
        } else {
            if (!this.onInsideExecuted) {
                if (!this.getOnInside().isEmpty()) {
                    this.getContext().logOnInside(this.getKielState());
                    this.getOnInside().execute();
                }
                this.onInsideExecuted = true;
            }
            return STRLReactingComponent.PAUSE;
        }
    }

    /**
     * This method is called if a superordinated cell is entered.
     * @param thc
     *            is ignored
     */
    public final void deepEnter(final boolean thc) {
        this.onInsideExecuted = false;
    }

    /**
     * Returns a list containing the corresponding kiel simple state.
     * @return Returns a list containing the corresponding kiel simple state.
     */
    public final ArrayList getActiveStates() {
        ArrayList result = new ArrayList();
        result.add(this.getKielState());
        return result;
    }

    /**
     * Prepares the state for the next instant.
     */
    public final void prepareForNext() {
        this.onInsideExecuted = false;
    }

    /**
     * Returns the set of potentially emitted signals. When this method is
     * called this simple state can be active.
     * @return Returns the set of potentially emitted signals.
     * @param fresh
     *            ignored
     * @param unknown
     *            ignored
     */
    public final Collection getRemainingPotential(final Collection fresh,
            final Collection unknown) {
        HashSet result = new HashSet();
        this.finalTest();
        if (!this.isFinal()) {
            if (!this.onInsideExecuted) {
                result.addAll(this.getOnInside().getPotential());
            }
        }
        return result;
    }

    /**
     * Sets the isreachedfinal flag to true, if the simple state is a final
     * state.
     */
    private void finalTest() {
        if (this.isFinal()) {
            this.isReachedFinal = true;
        }
    }

    /**
     * Nothing happens.
     */
    public final void deepExit() {
    }

    /**
     * nothing happens.
     * @param potential
     *            ignored
     */
    public final void update(final Collection potential) {
    }

    /**
     * nothing happens.
     */
    public final void finalUpdate() {
    }

    /**
     * Returns the set of potentially emitted signals. This method is called if
     * the simple state might become active.
     * @param fresh
     *            frische Signale
     * @param unknown
     *            unbekannte Signale
     * @return Gibt die Menge aller mÃ¶glicherweise noch emittierten Signale
     *         zurÃ¼ck.
     */
    public final Collection getCompletePotential(final Collection fresh,
            final Collection unknown) {
        this.finalTest();
        return this.getOnInside().getPotential();
    }

    /**
     * Returns an empty colleection.
     * @return Returns an empty colleection.
     */
    public final Collection getExitPotential() {
        return new HashSet();
    }

    /**
     * Returns the isReachedFinal flag.
     * @return Returns the isReachedFinal flag.
     */
    public final boolean reachesFinal() {
        return this.isReachedFinal;
    }

    /**
     * Resets the isReachedFinal flag.
     */
    public final void resetHelpingFlags() {
        this.isReachedFinal = false;
    }

    /**
     * Sets the context for the oninside actions.
     * @param sim
     *            context
     */
    public final void setComponentsContext(final STRLSimulator sim) {
        this.getOnInside().setContext(sim);
    }
}
```

.3.9 STRLSimulator.java

```
/*
 * Created on 30.06.2004
 */
package kiel.simulator.syncchart;

import java.io.PrintWriter;
import java.util.Collection;
import java.util.HashSet;
import java.util.Iterator;

import kiel.configMngr.Configuration;
import kiel.configMngr.ExecuteTransition;
import kiel.configMngr.MacroStep;
import kiel.configMngr.OnEntry;
import kiel.configMngr.OnExit;
import kiel.configMngr.OnInside;
import kiel.configMngr.SignalAbsent;
import kiel.configMngr.SignalPresent;
import kiel.configMngr.SignalUnknown;
import kiel.configMngr.SignalValue;
import kiel.configMngr.StateActivated;
import kiel.configMngr.StateDeactivated;
import kiel.configMngr.StateNotSuspended;
import kiel.configMngr.StateSuspended;
import kiel.configMngr.StateTransient;
import kiel.configMngr.TestTransition;
import kiel.configMngr.VariableValue;
import kiel.dataStructure.State;
import kiel.dataStructure.StateChart;
import kiel.dataStructure.Transition;
import kiel.dataStructure.Variable;
import kiel.dataStructure.eventexp.Event;
import kiel.dataStructure.eventexp.IntegerSignal;
import kiel.dataStructure.eventexp.Signal;
import kiel.simulator.Simulator;
import kiel.simulator.syncchart.action.STRLAction;
import kiel.simulator.syncchart.action.STRLDelayedActionHandler;
import kiel.simulator.syncchart.boolexp.STRLBooleanExpression;
import kiel.simulator.syncchart.converter.Context;
import kiel.simulator.syncchart.converter.KielToSyncChart;
import kiel.simulator.syncchart.exceptions.STRLException;
import kiel.simulator.syncchart.exceptions.STRLNotSupportedException;
import kiel.simulator.syncchart.sigexp.STRLSignal;
import kiel.util.LogFile;

/**
 * STRLSimulator controls the simulation.
 * @author Andre Ohlhoff
 */
public class STRLSimulator extends Context implements Simulator {

    /**
     * Feature name.
     */
    public static final String FEATUREHIDESUSPENDED
        = "HideSuspendedSubstates";

    /**
     * Feature. Status set by the user. If true, configurations will not contain
     * substates of suspended states.
     */
    private boolean featureHideSuspended = true;

    /**
     * is true, if a signal or a variable changes its status or value
     */
    private boolean somethingchanged = false;

    /**
     * a STRLDelayActionHandler holds all actions, which must be delayed.
     */
    private STRLDelayedActionHandler handler = new STRLDelayedActionHandler();

    /**
     * number of the current instant.
     */
    private int instant = 0;

    /**
     * the original kiel StateChart.
     */
    private StateChart kiel = null;

    /**
     * the internal SyncChart used to compute the reaction of the StateChart.
     */
    private SyncChart syncChart = null;

    /**
     * the current MacroStep.
     */
    private MacroStep macroStep = null;

    /**
     * Constructs a STRLSimulator.
     */
    public STRLSimulator() {
        this.handler.setContext(this);
        this.setLogFile(new LogFile(new PrintWriter(System.out)));
    }

    /**
     * Sets the StateChart.
     * @param sc kiel StateChart
     * @throws STRLException An exception is thrown, if the StateChart is not
     *             supported.
     */
    public final void setStateChart(final StateChart sc)
        throws STRLException {
        this.kiel = sc;
        this.clear();
        this.instant = 0;
        this.macroStep = new MacroStep(this.instant);
        KielToSyncChart kts = new KielToSyncChart(this);
        this.syncChart = kts.convert(this.kiel);

        this.syncChart.setContext(this);
        this.syncChart.initInputAndOutput();
        this.syncChart.activate();
    }

    /**
     * Resets the simulator.
     */
    public final void reset() {
        this.clear();
        this.instant = 0;
        this.macroStep = new MacroStep(this.instant);
        KielToSyncChart kts = new KielToSyncChart(this);
        try {
            this.syncChart = kts.convert(this.kiel);
            this.syncChart.setContext(this);
            this.syncChart.initInputAndOutput();
            this.syncChart.activate();
        } catch (STRLException e) {
            e.printStackTrace();
        }
    }

    /**
     * Computes the next step.
     * @return Returns a MacroStep as a summary of the reaction.
     * @throws STRLException The exception is thrown if something went wrong.
     */
    public final MacroStep nextStep() throws STRLException {

        int code = STRLReactingComponent.UNKNOWN;

        while (code == STRLReactingComponent.UNKNOWN) {

            this.somethingchanged = false;

            code = this.syncChart.react();

            this.handler.execute();

            Collection potential = this.syncChart.getRemainingPotential(
                    new HashSet(), new HashSet());
            potential.addAll(this.handler.getCompletePotential());

            this.showPotential(potential);

            this.syncChart.update(potential);

            if (code == STRLReactingComponent.UNKNOWN
                    && !this.somethingchanged) {
                throw this.createException(potential);
            }

        }

        boolean test = false;

        while (!test) {
            this.somethingchanged = false;
            // test if some valued signals emit are delayed
            test = this.handler.execute();
            Collection potential = this.handler.getCompletePotential();
            this.syncChart.update(potential);

            if (!this.somethingchanged && !test) {
                throw this.createException(potential);
            }
        }

        if (code == STRLReactingComponent.DEAD) {
            this.warning("syncchart␣returned␣dead");
        }

        this.syncChart.finalUpdate();

        this.logSim("end␣of␣instant:␣"
                + this.instant
                + "␣in:␣"
                + new Configuration(this.syncChart.getActiveStates())
                        .toString());
        this.instant++;
        MacroStep result = this.macroStep;
        this.macroStep = new MacroStep(this.instant);

        this.syncChart.prepareForNext();
        return result;
    }

    /**
     * Returns the input signals.
     * @return Returns the input signals.
     */
    public final Collection getInput() {
        return this.kiel.getInputEvents();
    }

    /**
     * Returns the output signals.
     * @return Returns the output signals.
     */
    public final Collection getOutput() {
        return this.kiel.getOutputEvents();
    }

    /**
     * Emits the event (which must be instanceof Signal) in the next instant.
     * @param e event to emit
     * @throws STRLException The exception is thrown if something went wrong.
     */
    public final void emit(final Event e) throws STRLException {
        if ((e instanceof Signal) && !(e instanceof IntegerSignal)) {
            this.syncChart.emit((Signal) e);
        } else {
            throw new STRLNotSupportedException("cannot␣emit␣" + e);
        }
    }
    /**
     * Emits the event (which must be instanceof IntegerSignal) with the given
     * value in the next instant.
     * @param e event to emit
     * @param i value
     * @throws STRLException The exception is thrown if something went wrong.
     */
    public final void emit(final Event e, final int i)
        throws STRLException {
        if (e instanceof IntegerSignal) {
            this.syncChart.emit((IntegerSignal) e, i);
        } else {
            throw new STRLNotSupportedException("cannot␣emit␣" + e
                    + "␣with␣a␣value");
        }
    }

    /**
     * Sets the somethingchanged flag to true.
     */
    public final void somethingChanged() {
        this.somethingchanged = true;
    }

    /**
     * Creates an exception saying that no status or value can be derived for
     * the signals in the given collection.
     * @param signals signals without status or valid value
     * @return The generated exception.
     */
    private STRLException createException(final Collection signals) {
        StringBuffer message = new StringBuffer();
        message.append("cannot␣derive␣state␣and/or␣value␣for:");
        Iterator iter = signals.iterator();
        while (iter.hasNext()) {
            message.append("␣");
            message.append(
                    ((STRLSignal) iter.next()).getKielEvent().getName());
        }
        return new STRLException(message.toString());
    }

    /**
     * Tries to execute an action. If the action cannout be executed, the
     * delayedaction handler delays the action until it can be executed.
     * @param action action to execute or delay
     * @throws STRLException The exception is thrown if something went wrong.
     */
    public final void tryExecute(final STRLAction action)
        throws STRLException {
        this.handler.tryExecute(action);
    }

    /**
     * Generates a logfile entry containing the current potential.
     * @param potential current potential
     */
    private void showPotential(final Collection potential) {
        Iterator iter = potential.iterator();
        StringBuffer s = new StringBuffer();
        s.append("potential:");
        while (iter.hasNext()) {
            s.append("␣");
            s.append(((STRLSignal) iter.next()).getKielEvent().getName());
        }
        this.debug(s.toString());
    }

    /**
     * Evaluates the given guard.
     * @param guard Guard to evaluate
     * @param fresh fresh signals
     * @param unknown unknown signals
     * @return Returns TRUE, FALSE or UNKNOWN.
     */
    public final int evaluateGuard(final STRLBooleanExpression guard,
            final Collection fresh, final Collection unknown) {
        return this.handler.evaluateGuard(guard, fresh, unknown);
    }

    /**
     * Generates a logfile entry if a signal must be emitted.
     * @param kielSignal kiel Signal
     */
    public final void logPresent(final Signal kielSignal) {
        this.somethingChanged();
        SignalPresent sp = new SignalPresent(kielSignal);
        this.logSim(sp.toString());
        this.macroStep.addMicroStep(sp);
    }

    /**
     * Generates a logfile entry if a signal cannot be emitted.
     * @param kielSignal kiel Signal
     */
    public final void logAbsent(final Signal kielSignal) {
        this.somethingChanged();
        SignalAbsent sa = new SignalAbsent(kielSignal);
        this.logSim(sa.toString());
        this.macroStep.addMicroStep(sa);
    }

    /**
     * Generates a logfile entry if a signal is unknown.
     * @param kielSignal kiel Signal
     */
    public final void logUnknown(final Signal kielSignal) {
        SignalUnknown sa = new SignalUnknown(kielSignal);
        this.logSim(sa.toString());
        this.macroStep.addMicroStep(sa);
    }

    /**
     * Generates a logfile entry if a transition is tested.
     * @param t transition that is tested.
     */
    public final void logTestTransition(final Transition t) {
        TestTransition tt = new TestTransition(t);
        this.logSim(tt.toString());
```

```java
        this.macroStep.addMicroStep(tt);
    }

    /**
     * Generates a logfile entry if a transition is executed.
     * @param t transition that is executed.
     */
    public final void logExecuteTransition(final Transition t) {
        ExecuteTransition et = new ExecuteTransition(t);
        this.logSim(et.toString());
        this.macroStep.addMicroStep(et);
    }

    /**
     * Generates a logfile entry if a state is activated.
     * @param state state that is activated.
     */
    public final void logActivate(final State state) {
        Configuration c = new Configuration(this.syncChart.getActiveStates());
        StateActivated sa = new StateActivated(state, c);
        this.logSim(sa.toString());
        this.macroStep.addMicroStep(sa);
    }

    /**
     * Generates a logfile entry if a state is deactivated.
     * @param state state that is deactivated.
     */
    public final void logDeactivate(final State state) {
        StateDeactivated sd = new StateDeactivated(state);
        this.logSim(sd.toString());
        this.macroStep.addMicroStep(sd);
    }

    /**
     * Generates a logfile entry if a state executes its onentry actions.
     * @param state onentry state
     */
    public final void logOnEntry(final State state) {
        OnEntry oe = new OnEntry(state);
        this.logSim(oe.toString());
        this.macroStep.addMicroStep(oe);
    }

    /**
     * Generates a logfile entry if a state executes its onexit actions.
     * @param state onexit state
     */
    public final void logOnExit(final State state) {
        OnExit oe = new OnExit(state);
        this.logSim(oe.toString());
        this.macroStep.addMicroStep(oe);
    }

    /**
     * Generates a logfile entry if a state executes its oninside actions.
     * @param state oninside state
     */
    public final void logOnInside(final State state) {
        OnInside oi = new OnInside(state);
        this.logSim(oi.toString());
        this.macroStep.addMicroStep(oi);
    }

    /**
     * Generates a logfile entry if a state is transient.
     * @param state transient state
     */
    public final void logTransient(final State state) {
        StateTransient st = new StateTransient(state);
        this.logSim(st.toString());
        this.macroStep.addMicroStep(st);
    }

    /**
     * Generates a logfile entry if a state is suspended.
     * @param state suspended state
     */
    public final void logSuspended(final State state) {
        Configuration c = new Configuration(this.syncChart.getActiveStates());
        StateSuspended ss = new StateSuspended(state, c);
        this.logSim(ss.toString());
        this.macroStep.addMicroStep(ss);
    }

    /**
     * Generates a logfile entry if a state is suspended anymore.
     * @param state not suspended state
     */
    public final void logNotSuspended(final State state) {
        if (this.featureHideSuspended) {
            Configuration c = new Configuration(this.syncChart
                    .getActiveStates());
            StateNotSuspended sns = new StateNotSuspended(state, c);
            this.logSim(sns.toString());
            this.macroStep.addMicroStep(sns);
        }
    }

    /**
     * Generates a logfile entry that a signal has a specific value.
     * @param signal kiel IntegerSignal
     * @param value value
     */
    public final void logSigValue(final IntegerSignal signal,
            final Integer value) {
        this.somethingChanged();
        SignalValue sv = new SignalValue(signal, value);
        this.logSim(sv.toString());
        this.macroStep.addMicroStep(sv);
    }

    /**
     * Generates a logfile entry that a variable has a specific value.
     * @param var kiel Variable
     * @param value value
     */
    public final void logVarValue(final Variable var, final Integer value) {
        this.somethingChanged();
        VariableValue vv = new VariableValue(var, value);
        this.logSim(vv.toString());
        this.macroStep.addMicroStep(vv);
    }

    /**
     * Sets the status of the given feature.
     * @param feature The feature name.
     * @param status The status of the feature.
     * @throws STRLNotSupportedException When a feature is not supported.
     */
    public final void setFeature(final String feature,
            final boolean status)
            throws STRLNotSupportedException {
        if (feature.equals(STRLSimulator.FEATUREHIDESUSPENDED)) {
            this.featureHideSuspended = status;
        } else {
            throw new STRLNotSupportedException("Feature␣" + feature
                    + "␣not␣supported");
        }
    }

    /**
     * Returns the status of the given feature.
     * @param feature The feature name.
     * @return Returns the status of the given feature.
     * @throws STRLNotSupportedException When a feature is not supported.
     */
    public final boolean getFeature(final String feature)
            throws STRLNotSupportedException {
        if (feature.equals(STRLSimulator.FEATUREHIDESUSPENDED)) {
            return this.featureHideSuspended;
        } else {
            throw new STRLNotSupportedException("Feature␣" + feature
                    + "␣not␣supported");

        }
    }

    /**
     * The syncChart simulator does not allow to set the values of variables.
     * @param value the value
     * @param var the variable
     * @throws STRLNotSupportedException This exception is thrown every time
     *              this method is called.
     */
    public final void setVariable(final Variable var, final String value)
            throws STRLNotSupportedException {
        throw new STRLNotSupportedException("Setting␣the␣value␣of␣a␣variable"
                + "is␣not␣supported␣by␣this␣simulator.");
    }

}
```

.3.10 STRLState.java

```
/*
 * Created on 14.05.2004
 */
package kiel.simulator.syncchart;

import kiel.dataStructure.State;
import kiel.simulator.syncchart.action.STRLActions;

/**
 * STRLState is the basic class for STRLMacroState and STRLSimpleState.
 * @author Andre Ohlhoff
 */
public abstract class STRLState extends STRLReactingComponent {

    /**
     * is true, if the state is active.
     */
    private boolean isActive = false;

    /**
     * is true, if the state was active before.
     */
    private boolean wasActiveBefore = false;

    /**
     * is true, if this state is final.
     */
    private boolean isFinal = false;

    /**
     * is true, if the surrounding cell has a deephistory connector.
     */
    private boolean hasDeepHistory = false;

    /**
     * oninside actions are executed if the state is active.
     */
    private STRLActions onInside = new STRLActions();

    /**
     * Reference to the corresponding kiel state.
     */
    private State kielState;

    /**
     * Constructs a STRLState with a reference to the corresponding kiel state.
     * @param state
     *            kiel state
     */
    public STRLState(final State state) {
        this.kielState = state;
    }

    /**
     * Sets the isActive flag.
     * @param active
```

```
 *            true if the state is active Flag
 */
private void setActive(final boolean active) {
    this.isActive = active;
}

/**
 * Returns true if the state is active.
 * @return Returns true if the state is active.
 */
public final boolean isActive() {
    return this.isActive;
}

/**
 * Sets the isActive flag to false, and if it was true, this wasActiveBefore
 * flag is set to true.
 */
public final void deactivate() {
    if ((!this.wasActiveBefore()) && this.isActive()) {
        this.wasActiveBefore = true;
    }
    this.getContext().logDeactivate(this.getKielState());
    this.setActive(false);
}

/**
 * Returns true, if the state was active some instant before.
 * @return Returns true, if the state was active some instant before.
 */
public final boolean wasActiveBefore() {
    return this.wasActiveBefore;
}

/**
 * Sets the isActive flag to true.
 */
public final void activate() {
    this.setActive(true);
    this.getContext().logActivate(this.getKielState());
}

/**
 * Returns true, if the surounding cell is entered through a deephistory.
 * @return Returns true, if the surounding cell is entered through a
 *         deephistory
 */
public final boolean hasDeepHistory() {
    return this.hasDeepHistory;
}

/**
 * Sets the hasdeephistory flag.
 * @param history
 *            true or false
 */
```

```
    public final void setDeepHistory(final boolean history) {
        this.hasDeepHistory = history;
    }

    /**
     * Sets the onside actions.
     * @param actions
     *            oninside actions OnInside Aktionen
     */
    public final void setOnInside(final STRLActions actions) {
        this.onInside = actions;
    }

    /**
     * Returns the onside actions.
     * @return Returns the onside actions.
     */
    public final STRLActions getOnInside() {
        return this.onInside;
    }

    /**
     * Returns true if this state is final.
     * @return Returns true if this state is final.
     */
    public final boolean isFinal() {
        return this.isFinal;
    }

    /**
     * Sets the isfinal flag to true.
     */
    public final void setFinal() {
        this.isFinal = true;
    }

    /**
     * Returns the corresponding kiel state.
     * @return Returns the corresponding kiel state.
     */
    public final State getKielState() {
        return this.kielState;
    }

}
```

.3.11 STRLStateCell.java

```java
/*
 * Created on 13.06.2004
 */
package kiel.simulator.syncchart;

import java.util.ArrayList;
import java.util.Collection;
import java.util.HashSet;

import kiel.dataStructure.State;
import kiel.simulator.syncchart.action.STRLActions;
import kiel.simulator.syncchart.boolexp.STRLBooleanExpression;
import kiel.simulator.syncchart.exceptions.STRLException;
import kiel.simulator.syncchart.exceptions.STRLNotSupportedException;
import kiel.simulator.syncchart.transition.STRLNormalTermination;
import kiel.simulator.syncchart.transition.STRLStrongAbortion;
import kiel.simulator.syncchart.transition.STRLTransitionHandler;
import kiel.simulator.syncchart.transition.STRLWeakAbortion;

/**
 * STRLStateCell is the most important class.
 * @author Andre Ohlhoff
 */
public class STRLStateCell extends STRLReactiveCell {

    /**
     * this constant is used as position marker, showing that the cell will
     * initialize its strong abortion transitions next.
     */
    private static final int INITSA = 0;

    /**
     * this constant is used as position marker, showing that the cell is
     * checking its strongabortion transitions.
     */
    private static final int CHECKSA = 1;

    /**
     * this constant is used as position marker, showing that the cell is
     * checking its onentry actions.
     */
    private static final int CHECKENTRY = 2;

    /**
     * this constant is used as position marker, showing that the cell is
     * checking the suspension.
     */
    private static final int CHECKSUSP = 3;

    /**
     * this constant is used as position marker, showing that the cell is
     * executing its body.
     */
    private static final int EXECUTEBODY = 4;

    /**
     * this constant is used as position marker, showing that the cell is
     * checking its weakabortion transitions.
     */
    private static final int CHECKWA = 6;

    /**
     * this constant is used as position marker, showing that the cell is
     * checking its normal termination transitions.
     */
    private static final int CHECKNT = 7;

    /**
     * this constant is used as position marker, showing that the cell is
     * exitting.
     */
    private static final int EXIT = 8;

    /**
     * is true, if the cell was entered in this instant.
     */
    private boolean isFirstInstant = true;

    /**
     * stores the return code of the body to avoid redundant computations.
     */
    private int bodyCode = STRLReactingComponent.UNKNOWN;

    /**
     * shows what the cell is doing now or next.
     */
    private int position = STRLStateCell.CHECKSA;

    /**
     * transitionhandler with an enabled transition used later to exit the cell.
     */
    private STRLTransitionHandler exitHandler = null;

    /**
     * body state of the cell.
     */
    private STRLState body = null;

    /**
     * transition handler for the strongabortion transitions.
     */
    private STRLTransitionHandler strongAbort = new STRLTransitionHandler();

    /**
     * transition handler for the weakabortion transitions.
     */
    private STRLTransitionHandler weakAbort = new STRLTransitionHandler();

    /**
     * transition handler for the normaltermination transitions.
     */
    private STRLTransitionHandler normTerm = new STRLTransitionHandler();
    /**
     * STRLSuspend to suspend the reaction of the body.
     */
    private STRLSuspend suspend = new STRLSuspend();

    /**
     * STRLActions executed when the cell is entered.
     */
    private STRLActions onEntry = new STRLActions();

    /**
     * STRLActions executed when the cell is exited.
     */
    private STRLActions onExit = new STRLActions();

    /**
     * True if the body of the cell is suspended this instant.
     */
    private boolean isSuspended = false;

    /**
     * True if the body of the cell is transient.
     */
    private boolean isTransient = false;

    /**
     * True, if the cell is reached during computation of the potential.
     */
    private boolean isPossibleActive = false;

    /**
     * Constructs a new STRLStateCell for a kiel State.
     * @param state kiel State.
     */
    public STRLStateCell(final State state) {
        super(state);
    }

    /**
     * Sets the body of the cell.
     * @param b body of the cell.
     */
    public final void setBody(final STRLState b) {
        this.body = b;
    }

    /**
     * Sets the context for all components.
     * @param sim STRLSimulator
     */
    public final void setComponentsContext(final STRLSimulator sim) {
        this.strongAbort.setContext(sim);
        this.weakAbort.setContext(sim);
        this.normTerm.setContext(sim);
        this.body.setContext(sim);
        this.onEntry.setContext(sim);
        this.onExit.setContext(sim);
    }

    /**
     * Sets the strongabortion transitions.
     * @param aborts strongabortion transitions
     */
    public final void setStrongAborts(final STRLStrongAbortion[] aborts) {
        this.strongAbort.setTransitions(aborts);
    }

    /**
     * Sets the weakabortion transitions.
     * @param aborts weakabortion transitions
     */
    public final void setWeakAborts(final STRLWeakAbortion[] aborts) {
        this.weakAbort.setTransitions(aborts);
    }

    /**
     * Sets the normaltermination transitions.
     * @param terms normaltermination transitions.
     */
    public final void setNormTerms(final STRLNormalTermination[] terms) {
        this.normTerm.setTransitions(terms);
    }

    /**
     * Sets the STRLSuspend.
     * @param s STRLSuspend
     */
    public final void setSuspension(final STRLSuspend s) {
        this.suspend = s;
    }

    /**
     * Sets the onentry actions.
     * @param a onentry actions.
     */
    public final void setOnEntry(final STRLActions a) {
        this.onEntry = a;
    }

    /**
     * Sets the onexit actions.
     * @param a onexit actions.
     */
    public final void setOnExit(final STRLActions a) {
        this.onExit = a;
    }

    /**
     * Starts the reaction.
     * @return Returns DEAD, DONE, PAUSE or UNKNOWN
     * @throws STRLException The exception is thrown if something went wrong.
     */
    public final int react() throws STRLException {
        switch (this.position) {
        case STRLStateCell.INITSA:
            return this.resetSA();
        case STRLStateCell.CHECKSA:
            return this.checkSA();
        case STRLStateCell.CHECKENTRY:
```

```
        return this.checkEntry();
    case STRLStateCell.CHECKSUSP:
        return this.checkSusp();
    case STRLStateCell.EXECUTEBODY:
        return this.executeBody();
    case STRLStateCell.CHECKWA:
        return this.checkWA();
    case STRLStateCell.CHECKNT:
        return this.checkNT();
    case STRLStateCell.EXIT:
        return this.exit();
    default: // dead code ;-)
        throw new STRLException(
                "simulator␣error:␣unknown␣position␣in␣StateCell.");
    }

}

/**
 * Initializes the strongabortion transitions and then checks if a
 * strongabortion transition must be enabled.
 * @return Returns DEAD, DONE, PAUSE or UNKNOWN
 * @throws STRLException The exception is thrown if something went wrong.
 */
private int resetSA() throws STRLException {
    if (this.isFirstInstant) {
        this.strongAbort.init();
    }
    this.position = STRLStateCell.CHECKSA;
    return this.checkSA();
}

/**
 * Initializes the weakabortion transitions and then checks if a
 * weakabortion transition must be enabled.
 * @return Returns DEAD, DONE, PAUSE or UNKNOWN
 * @throws STRLException The exception is thrown if something went wrong.
 */
private int resetWA() throws STRLException {
    if (this.isFirstInstant) {
        this.weakAbort.init();
    }
    this.position = STRLStateCell.CHECKWA;
    return this.checkWA();
}

/**
 * Initializes the normaltermination transitions and then checks if a
 * normaltermination transition must be enabled.
 * @return Returns DEAD, DONE, PAUSE or UNKNOWN
 * @throws STRLException The exception is thrown if something went wrong.
 */
private int resetNT() throws STRLException {
    if (this.isFirstInstant) {
        this.normTerm.init();
    }
    this.position = STRLStateCell.CHECKNT;
    return this.checkNT();
}
```

```
/**
 * Checks if a strongabortion transition must be enabled. If no transition
 * can be enabled, the onentry actions are checked.
 * @return Returns DEAD, DONE, PAUSE or UNKNOWN
 * @throws STRLException The exception is thrown if something went wrong.
 */
private int checkSA() throws STRLException {
    int check = this.strongAbort.evaluate(this.isFirstInstant);
    if (check == STRLTransitionHandler.UNKNOWN) {
        return STRLReactingComponent.UNKNOWN;
    } else if (check == STRLTransitionHandler.FALSE) {
        this.position = STRLStateCell.CHECKENTRY;
        return this.checkEntry();
    } else {
        /* check == STRLTransitionHandler.TRUE */
        if (this.isFirstInstant) {
            this.isTransient = true;
        }
        this.exitHandler = this.strongAbort;
        this.position = STRLStateCell.EXIT;
        return this.exit();
    }
}

/**
 * Executes the onentry actions, if firstinstant is true. Next the
 * STRLSuspend is checked.
 * @return Returns DEAD, DONE, PAUSE or UNKNOWN
 * @throws STRLException The exception is thrown if something went wrong.
 */
private int checkEntry() throws STRLException {
    if (this.isFirstInstant) {
        this.body.activate();
        if (!this.onEntry.isEmpty()) {
            this.getContext().logOnEntry(this.body.getKielState());
            this.onEntry.execute();
        }
    }
    this.position = STRLStateCell.CHECKSUSP;
    return this.checkSusp();
}

/**
 * Exits the cell and executes the effect of the enabled transition.
 * @return Returns DEAD, DONE, PAUSE or UNKNOWN
 * @throws STRLException The exception is thrown if something went wrong.
 */
private int exit() throws STRLException {
    if (!this.isTransient) {
        this.deepExit();
    } else {
        this.getContext().logTransient(this.body.getKielState());
    }
    this.setNextCell(this.exitHandler.nextCell());
    this.exitHandler.execute();
    return STRLReactingComponent.DONE;
}

/**
 * Checks if the body must be suspended.
```

```
     * @return Returns DEAD, DONE, PAUSE or UNKNOWN
     * @throws STRLException The exception is thrown if something went wrong.
     */
    private int checkSusp() throws STRLException {
        int check = this.suspend.evaluate(this.isFirstInstant);
        if (check == STRLBooleanExpression.TRUE) {
            this.bodyCode = STRLReactingComponent.PAUSE;
            this.isSuspended = true;
            this.getContext().logSuspended(this.body.getKielState());
            return this.resetWA();
        } else if (check == STRLBooleanExpression.FALSE) {
            if (this.isSuspended) {
                this.isSuspended = false;
                this.getContext().logNotSuspended(this.body.getKielState());
            } else {
                this.isSuspended = false;
            }
            this.position = STRLStateCell.EXECUTEBODY;
            return this.executeBody();
        } else {
            /* check == STRLDelayExpression.UNKNOWN */
            return STRLReactingComponent.UNKNOWN;
        }
    }

    /**
     * Executes the body.
     * @return Returns DEAD, DONE, PAUSE or UNKNOWN
     * @throws STRLException The exception is thrown if something went wrong.
     */
    private int executeBody() throws STRLException {
        this.bodyCode = this.body.react();
        if (this.bodyCode == STRLReactingComponent.UNKNOWN) {
            return this.bodyCode;
        } else {
            return this.resetWA();
        }
    }

    /**
     * Checks if a weakabortion transition must be enabled. If no transition can
     * be enabled, the normaltermination transitions are checked.
     * @return Returns DEAD, DONE, PAUSE or UNKNOWN
     * @throws STRLException The exception is thrown if something went wrong.
     */
    private int checkWA() throws STRLException {
        int check = this.weakAbort.evaluate(this.isFirstInstant);
        if (check == STRLTransitionHandler.TRUE) {
            this.exitHandler = this.weakAbort;
            this.position = STRLStateCell.EXIT;
            return this.exit();
        } else if (check == STRLTransitionHandler.FALSE) {
            return this.resetNT();
        } else {
            /* check == STRLTransitionHandler.UNKNOWN */
            return STRLReactingComponent.UNKNOWN;
        }
    }

    /**
     * Checks if a normaltermination transition must be enabled. If no
     * transition can be enabled, PAUSE is returned.
     * @return Returns DEAD, DONE, PAUSE or UNKNOWN
     * @throws STRLException The exception is thrown if something went wrong.
     */
    private int checkNT() throws STRLException {
        if (this.bodyCode == STRLReactingComponent.DEAD) {
            int check = this.normTerm.evaluate(false);
            if (check == STRLTransitionHandler.TRUE) {
                this.exitHandler = this.normTerm;
                this.position = STRLStateCell.EXIT;
                return this.exit();
            } else if (check == STRLTransitionHandler.FALSE) {
                if (this.body.isFinal()) {
                    return STRLReactingComponent.DEAD;
                } else {
                    return STRLReactingComponent.PAUSE;
                }
            } else {
                /* check == STRLTransitionHandler.UNKNOWN */
                return STRLReactingComponent.UNKNOWN;
            }
        } else {
            return STRLReactingComponent.PAUSE;
        }
    }

    /**
     * Returns the list of active states.
     * @return Returns the list of active states.
     */
    public final ArrayList getActiveStates() {
        try {
            if (!this.getContext().getFeature(
                    STRLSimulator.FEATUREHIDESUSPENDED)
                    || !this.isSuspended) {
                return this.body.getActiveStates();
            } else {
                ArrayList result = new ArrayList();
                result.add(this.body.getKielState());
                return result;
            }
        } catch (STRLNotSupportedException e) {
            // dead code
            // feature is supported
            return null;
        }
    }

    /**
     * Prepares the state for the next instant.
     */
    public final void prepareForNext() {
        this.isFirstInstant = false;
        if (!this.isSuspended) {
            this.body.prepareForNext();
        }
        this.init();
    }
```

```java
/**
 * This method is called if an incoming tranition is enabled and executed.
 * @param thc is true, if a history connector is used.
 */
public final void enter(final boolean thc) {
    this.isFirstInstant = true;
    this.body.deepEnter(thc);
    this.isSuspended = false;
    this.init();
}

/**
 * This method is called if a superordinated cell is entered.
 * @param thc is true, if a history connector is used.
 */
public final void deepEnter(final boolean thc) {
    this.body.deepEnter(true);
    this.init();
}

/**
 * Initializes some members.
 */
private void init() {
    this.isPossibleActive = true;
    this.position = STRLStateCell.INITSA;
    this.strongAbort.prepareForNext();
    this.weakAbort.prepareForNext();
    this.normTerm.prepareForNext();
    this.isTransient = false;
    this.bodyCode = STRLReactingComponent.UNKNOWN;
    this.exitHandler = null;
}

/**
 * This method is called, if a superordinated cell is exited.
 * @throws STRLException The exception is thrown if something went wrong.
 */
public final void deepExit() throws STRLException {
    if (!this.body.hasDeepHistory()) {
        this.body.deepExit();
    }
    if (!onExit.isEmpty()) {
        this.getContext().logOnExit(this.body.getKielState());
        this.onExit.execute();
    }
    this.body.deactivate();
    this.isPossibleActive = false;
}

/**
 * Returns the remaining potential.
 * @param fresh fresh signals
 * @param unknown unknown signals
 * @return Returns the remaining potential.
 */
public final Collection getRemainingPotential(final Collection fresh,
        final Collection unknown) {

    this.isPossibleActive = !this.isFirstInstant;
```

```java
    Collection result = new HashSet();

    result.addAll(
            this.strongAbort.getPotential(this.isFirstInstant, fresh,
            unknown));

    if (this.strongAbort.isPossibleEnabled()) {
        result.addAll(this.getExitPotential());
    }

    if (this.strongAbort.isSureEnabled()) {
        return result;
    }

    this.getContext().debug(
            "possible␣active:␣" + this.getKielNode().getID()
                    + this.body.getKielState().getName());

    this.isPossibleActive = true;

    if (this.isFirstInstant
            && (this.position <= STRLStateCell.CHECKENTRY)) {
        result.addAll(this.onEntry.getPotential());
    }
    int test = this.suspend.evaluate(this.isFirstInstant);

    if (test != STRLBooleanExpression.TRUE) {
        result.addAll(this.body.getRemainingPotential(fresh, unknown));
    }

    result.addAll(
            this.weakAbort.getPotential(this.isFirstInstant, fresh,
            unknown));

    if (this.weakAbort.isPossibleEnabled()) {
        result.addAll(this.getExitPotential());
    }

    if (this.weakAbort.isSureEnabled()) {
        return result;
    }

    if (this.body.reachesFinal()) {

        result.addAll(this.normTerm.getPotential(false, fresh, unknown));

        if (this.normTerm.isPossibleEnabled()) {
            result.addAll(this.getExitPotential());
        }

    }

    return result;
}

/**
 * Returns the complete potential.
 * @param fresh fresh signals
 * @param unknown unknown signals
 * @return Returns the complete potential.
 */
public final Collection getCompletePotential(final Collection fresh,
        final Collection unknown) {

    this.resetHelpingFlags();

    Collection result = new HashSet();
    result.addAll(this.strongAbort.getPotential(true, fresh, unknown));

    if (this.strongAbort.isSureEnabled()) {
        return result;
    }

    this.getContext().debug(
            "possible␣active:␣" + this.getKielNode().getID()
                    + this.body.getKielState().getName());

    this.isPossibleActive = true;

    result.addAll(this.onEntry.getPotential());

    int test = this.suspend.evaluate(true, fresh, unknown);

    if (test != STRLBooleanExpression.TRUE) {
        result.addAll(this.body.getCompletePotential(fresh, unknown));
    }
    result.addAll(this.weakAbort.getPotential(true, fresh, unknown));
    if (this.weakAbort.isPossibleEnabled()) {
        result.addAll(this.getExitPotential());
    }

    if (this.weakAbort.isSureEnabled()) {
        this.reachesFinal();
        return result;
    }

    if (this.body.reachesFinal()) {
        result.addAll(this.normTerm.getPotential(false, fresh, unknown));
        if (this.normTerm.isPossibleEnabled()) {
            result.addAll(this.getExitPotential());
        }
    }

    return result;
}

/**
 * Returns the body of the cell.
 * @return Returns the body of the cell.
 */
public final STRLState getBody() {
    return this.body;
}

/**
 * Returns true, if the cell has a deeo history connector.
 * @return Returns true, if the cell has a deeo history connector.
 */
public final boolean hasDeepHistory() {
    return this.body.hasDeepHistory();
}

/**
 * Updates all local signals.
 * @param potential Set of potentially emitted signals.
 */
public final void update(final Collection potential) {
    this.body.update(potential);
}

/**
 * Finally updates also signals, which are not visible.
 */
public final void finalUpdate() {
    this.body.finalUpdate();
}

/**
 * Returns true, if during computation of the potential a final state is
 * reached.
 * @return Returns true, if during computation of the potential a final
 *         state is reached
 */
public final boolean reachesFinal() {
    return this.body.reachesFinal() && this.body.isFinal();
}

/**
 * Returns the exit potential.
 * @return Returns the exit potential.
 */
public final Collection getExitPotential() {
    HashSet result = new HashSet();
    if (this.isPossibleActive) {
        result.addAll(this.body.getExitPotential());
        result.addAll(this.onExit.getPotential());
        this.isPossibleActive = false;
    }
    return result;
}

/**
 * Resets some helping flags.
 */
public final void resetHelpingFlags() {
    this.body.resetHelpingFlags();
    this.isPossibleActive = false;
}

}
```

.3.12 STRLSTG.java

```
/*
 * Created on 14.05.2004
 */
package kiel.simulator.syncchart;

import java.util.ArrayList;
import java.util.Collection;
import java.util.HashSet;
import java.util.Iterator;

import kiel.simulator.syncchart.exceptions.STRLException;
import kiel.simulator.syncchart.intexp.STRLIntegerVariable;
import kiel.simulator.syncchart.intexp.STRLVariableHandler;

/**
 * STRLSTG implements the behavior of a basic state transition graph.
 * @author Andre Ohlhoff
 */
public class STRLSTG extends STRLReactingComponent {

    /**
     * constant used as position marker, showing that the stg must initialize
     * its local variables.
     */
    private static final int INITIALIZING = 0;

    /**
     * constant used as position marker, showing that the stg computes the
     * normal reaction.
     */
    private static final int REACTING = 1;

    /**
     * shows what the stg does now or next.
     */
    private int position = STRLSTG.INITIALIZING;

    /**
     * is true, if the stgs has reacted at least once.
     */
    private boolean reactedOnce = false;

    /**
     * the initial cell of the stg.
     */
    private STRLInitialCell initialCell = null;

    /**
     * the cell which is currently active.
     */
    private STRLReactiveCell currentCell = null;

    /**
     * all contained cells.
     */
    private Collection cells = new HashSet();

    /**
     * stores the last return code, to avoid redundant computations.
     */
    private int lastResult = STRLReactingComponent.UNKNOWN;

    /**
     * is true, if the corresponding kiel Region contains a deep history pseudo
     * state.
     */
    private boolean hasDeepHistory = false;

    /**
     * a STRLVariableHandler holds all local variables.
     */
    private STRLVariableHandler varHandler = new STRLVariableHandler();

    /**
     * Sets the local variables.
     * @param strlVars local variables.
     */
    public final void setVariables(final STRLIntegerVariable[] strlVars) {
        this.varHandler.setVariables(strlVars);
    }

    /**
     * Sets the cells in the stg.
     * @param c cells in the stg.
     */
    public final void setCells(final STRLReactiveCell[] c) {
        for (int i = 0; i < c.length; i++) {
            this.cells.add(c[i]);
            if (c[i] instanceof STRLInitialCell) {
                this.setInitialState(((STRLInitialCell) c[i]));
            }
        }
    }

    /**
     * Sets the initial cell.
     * @param cell initial cell
     */
    private void setInitialState(final STRLInitialCell cell) {
        this.initialCell = cell;
        this.currentCell = this.initialCell;
    }

    /**
     * Sets the hasdeephistory flag to true.
     */
    public final void setDeepHistory() {
        this.hasDeepHistory = true;
    }

    /**
     * Starts the reaction.
     * @return Return UNKNOWN, DEAD or PAUSE
     * @throws STRLException The exception is thrown if something went wrong.
     */
    public final int react() throws STRLException {
        this.reactedOnce = true;
        if (this.lastResult == STRLReactingComponent.UNKNOWN) {
            if (this.position == STRLSTG.INITIALIZING) {
                return this.initializing();
            } else {
                return this.executing();
            }
        } else {
            return this.lastResult;
        }
    }

    /**
     * Initializes the local variables and starts the normal reaction.
     * @return Return UNKNOWN, DEAD or PAUSE
     * @throws STRLException The exception is thrown if something went wrong.
     */
    private int initializing() throws STRLException {
        this.varHandler.initialize();
        this.position = STRLSTG.REACTING;
        return this.executing();
    }

    /**
     * Starts the reaction.
     * @return Return UNKNOWN, DEAD or PAUSE
     * @throws STRLException The exception is thrown if something went wrong.
     */
    private int executing() throws STRLException {

        if (this.lastResult == STRLReactingComponent.UNKNOWN) {
            do {
                this.lastResult = this.currentCell.react();
                this.getContext().debug(
                        "cell:␣" + this.currentCell.getKielNode().getID()
                                + "␣returned␣" + this.lastResult);
                if (this.lastResult == STRLReactingComponent.DONE) {
                    this.currentCell = this.currentCell.nextCell();
                }
            } while (this.lastResult == STRLReactingComponent.DONE);

        }

        return this.lastResult;

    }

    /**
     * Sets the context for all cells.
     * @param sim STRLSimulator as context.
     */
    public final void setComponentsContext(final STRLSimulator sim) {
        this.varHandler.setContext(sim);
        STRLComponent.setContext(sim, this.cells);
    }

    /**
     * This method is called, if a superordinated cell is exited.
     * @throws STRLException The exception is thrown if something went wrong.
     */
    public final void deepExit() throws STRLException {
        this.currentCell.deepExit();
    }

    /**
     * Returns the list of active states.
     * @return Returns the list of active states.
     */
    public final ArrayList getActiveStates() {
        return this.currentCell.getActiveStates();
    }

    /**
     * Returns the remaining potential.
     * @param fresh fresh signals
     * @param unknown unknown signals
     * @return Returns the remainig potential.
     */
    public final Collection getRemainingPotential(final Collection fresh,
            final Collection unknown) {
        if (this.lastResult == STRLReactingComponent.UNKNOWN) {
            return this.currentCell.getRemainingPotential(fresh, unknown);
        } else {
            return new HashSet();
        }
    }

    /**
     * Returns the complete potential.
     * @param fresh fresh signals
     * @param unknown unknown signals
     * @return Returns the complete potential.
     */
    public final Collection getCompletePotential(final Collection fresh,
            final Collection unknown) {
        if (this.hasDeepHistory()) {
            return this.currentCell.getCompletePotential(fresh, unknown);
        } else {
            return this.initialCell.getCompletePotential(fresh, unknown);
        }
    }

    /**
     * Prepares the stg for the next instant.
     */
    public final void prepareForNext() {
        this.position = STRLSTG.REACTING;
        this.lastResult = STRLReactingComponent.UNKNOWN;
        this.currentCell.prepareForNext();
    }

    /**
     * This method is called if a superordinated cell is entered.
     * @param thc is true, if a deep history connector is used
     */
    public final void deepEnter(final boolean thc) {
```

```java
        if (thc && this.reactedOnce) {
            this.position = STRLSTG.REACTING;
        } else {
            this.position = STRLSTG.INITIALIZING;
            this.currentCell = this.initialCell;
        }

        this.lastResult = STRLReactingComponent.UNKNOWN;

        this.currentCell.deepEnter(thc);
    }

    /**
     * Returns true, if the corresponding kiel region contains a deep history
     * pseudo state.
     * @return Returns true, if the corresponding kiel region contains a deep
     *         history pseudo state.
     */
    public final boolean hasDeepHistory() {
        return this.hasDeepHistory;
    }

    /**
     * Updates the local signals in the current cell.
     * @param potential Updates the local signals in the current cell.
     */
    public final void update(final Collection potential) {
        this.currentCell.update(potential);
    }

    /**
     * Finally updates the signals in all cells.
     */
    public final void finalUpdate() {
        Iterator iter = this.cells.iterator();
        while (iter.hasNext()) {
            ((STRLReactiveCell) iter.next()).finalUpdate();
        }
    }

    /**
     * Returns true, if a final state is reached during the computation of the
     * potential.
     * @return Returns true, if a final state is reached during the computation
     *         of the potential.
     */
    public final boolean reachesFinal() {
        boolean result = false;
        Iterator iter = this.cells.iterator();
        while (iter.hasNext()) {
            result =
                ((STRLReactiveCell) iter.next()).reachesFinal() || result;
        }
        return result;
    }

    /**
     * Returns the exit potential of the stg.
     * @return Returns the exit potential of the stg.
     */
    public final Collection getExitPotential() {
        HashSet result = new HashSet();
        Iterator iter = this.cells.iterator();
        while (iter.hasNext()) {
            result.addAll(
                    ((STRLReactiveCell) iter.next()).getExitPotential());
        }
        return result;
    }

    /**
     * Resets some helping flags.
     */
    public final void resetHelpingFlags() {
        Iterator iter = this.cells.iterator();
        while (iter.hasNext()) {
            ((STRLReactiveCell) iter.next()).resetHelpingFlags();
        }
    }

}
```

.3.13 STRLSTGHandler.java

```
/*
 * Created on 17.09.2004
 */
package kiel.simulator.syncchart;

import java.util.ArrayList;
import java.util.Collection;
import java.util.HashSet;
import java.util.Iterator;
import java.util.LinkedList;

import kiel.simulator.syncchart.exceptions.STRLException;

/**
 * A STRLSTGHandler holds the STRLSTGs for a STRLMacroState.
 * @author Andre Ohlhoff
 */
public class STRLSTGHandler extends STRLReactingComponent {

    /**
     * list of stgs.
     */
    private LinkedList stgs = new LinkedList();

    /**
     * Sets the stgs.
     * @param s
     *            STRLSTGs
     */
    public final void setSTGs(final STRLSTG[] s) {
        for (int i = 0; i < s.length; i++) {
            this.stgs.add(s[i]);
        }
    }

    /**
     * Prepares all stgs for the next instant.
     */
    public final void prepareForNext() {
        Iterator iter = this.stgs.iterator();
        while (iter.hasNext()) {
            ((STRLSTG) iter.next()).prepareForNext();
        }
    }

    /**
     * Starts the reaction. Every stg reacts and the maximum of the return
     * values is returned.
     * @return Returns the maximum of the return values of the stgs reactions.
     * @throws STRLException
     *             The exception is thrown if something went wrong.
     */
    public final int react() throws STRLException {

        int maximum = STRLReactingComponent.DEAD;
        Iterator iter = this.stgs.iterator();

        while (iter.hasNext()) {
            STRLSTG next = (STRLSTG) iter.next();
            int temp = next.react();
            if (temp > maximum) {
                maximum = temp;
            }
        }
        return maximum;
    }

    /**
     * Sets the context for the stgs.
     * @param sim
     *            STRLSimulator
     */
    public final void setComponentsContext(final STRLSimulator sim) {
        STRLComponent.setContext(sim, this.stgs);
    }

    /**
     * Returns the union of active states from every stg.
     * @return Returns the union of active states from every stg.
     */
    public final ArrayList getActiveStates() {
        ArrayList states = new ArrayList();
        Iterator iter = this.stgs.iterator();
        while (iter.hasNext()) {
            states.addAll(((STRLSTG) iter.next()).getActiveStates());
        }
        return states;
    }

    /**
     * Returns the remaining potential.
     * @param fresh
     *            fresh signals
     * @param unknown
     *            unknown signals
     * @return Returns the remaining potential.
     */
    public final Collection getRemainingPotential(final Collection fresh,
            final Collection unknown) {
        HashSet result = new HashSet();
        Iterator iter = this.stgs.iterator();
        while (iter.hasNext()) {
            result.addAll(((STRLSTG) iter.next()).getRemainingPotential(fresh,
                    unknown));
        }
        return result;
    }

    /**
     * This method is called if a superordinated cell is entered.
     * @param thc
 *             is true, if a deep history connector is used.
 */
public final void deepEnter(final boolean thc) {
    Iterator iter = this.stgs.iterator();
    while (iter.hasNext()) {
        ((STRLSTG) iter.next()).deepEnter(thc);
    }
}

/**
 * This method is called if a superordinated cell is exited.
 * @throws STRLException
 *             The exception is thrown if something went wrong.
 */
public final void deepExit() throws STRLException {
    Iterator iter = this.stgs.iterator();
    while (iter.hasNext()) {
        ((STRLSTG) iter.next()).deepExit();
    }
}

/**
 * Updates the local signals in all stgs.
 * @param potential
 *             Set of potentially emitted signals.
 */
public final void update(final Collection potential) {
    Iterator iter = this.stgs.iterator();
    while (iter.hasNext()) {
        ((STRLSTG) iter.next()).update(potential);
    }
}

/**
 * Finally updates all local signals.
 */
public final void finalUpdate() {
    Iterator iter = this.stgs.iterator();
    while (iter.hasNext()) {
        ((STRLSTG) iter.next()).finalUpdate();
    }
}

/**
 * Returns the complete potential.
 * @param fresh
 *             fresh signals
 * @param unknown
 *             unknown signals
 * @return Returns the complete potential.
 */
public final Collection getCompletePotential(final Collection fresh,
        final Collection unknown) {
    HashSet result = new HashSet();
    Iterator iter = this.stgs.iterator();
    while (iter.hasNext()) {
        result.addAll(((STRLSTG) iter.next()).getCompletePotential(fresh,
                unknown));
    }
    return result;
}

/**
 * Returns true, if all stg can reach a final state during computation of
 * the potential.
 * @return Returns true, if all stg can reach a final state during
 *         computation of the potential.
 */
public final boolean reachesFinal() {
    boolean result = true;
    Iterator iter = this.stgs.iterator();
    while (iter.hasNext()) {
        result = ((STRLSTG) iter.next()).reachesFinal() && result;
    }
    return result;
}

/**
 * Returns the exit potential.
 * @return Returns the exit potential.
 */
public final Collection getExitPotential() {
    HashSet result = new HashSet();
    Iterator iter = this.stgs.iterator();
    while (iter.hasNext()) {
        result.addAll(((STRLSTG) iter.next()).getExitPotential());
    }
    return result;
}

/**
 * Resets some helping flags.
 */
public final void resetHelpingFlags() {
    Iterator iter = this.stgs.iterator();
    while (iter.hasNext()) {
        ((STRLSTG) iter.next()).resetHelpingFlags();
    }
}

}
```

.3.14 STRLSuspend.java

```java
/*
 * Created on 22.10.2004
 */
package kiel.simulator.syncchart;

import java.util.Collection;
import java.util.HashSet;

import kiel.simulator.syncchart.boolexp.STRLBooleanExpression;
import kiel.simulator.syncchart.boolexp.STRLFalse;

/**
 * A STRLSuspend contains an isImmediate flag and a signal expression.
 * @author Andre Ohlhoff
 */
public class STRLSuspend {

    /**
     * True, if the signal expression is evaluated immediate after entry of the
     * cell.
     */
    private boolean isImmediate = false;

    /**
     * Signal expression as trigger.
     */
    private STRLBooleanExpression signalExpression = new STRLFalse();

    /**
     * Sets the isImmediate flag to true.
     */
    public final void setImmediate() {
        this.isImmediate = true;
    }

    /**
     * Sets the signal expression.
     * @param sigExp
     *            the trigger
     */
    public final void setSigExp(final STRLBooleanExpression sigExp) {
        this.signalExpression = sigExp;
    }

    /**
     * Evaluates ths suspend.
     * @param firstInstant
     *            true, if the cell is activated this instant.
     * @param fresh
     *            fresh signals
     * @param unknown
     *            unknown signals
     * @return FALSE, TRUE or UNKNOWN
     */
    public final int evaluate(final boolean firstInstant,
            final Collection fresh, final Collection unknown) {
        if ((!this.isImmediate) && firstInstant) {
            return STRLBooleanExpression.FALSE;
        } else {
            return this.signalExpression.evaluate(fresh, unknown);
        }
    }

    /**
     * Evaluates the suspend.
     * @param firstInstant
     *            true, if the cell is activated this instant.
     * @return FALSE, TRUE or UNKNOWN
     */
    public final int evaluate(final boolean firstInstant) {
        return this.evaluate(firstInstant, new HashSet(), new HashSet());
    }
}
```

.3.15 SyncChart.java

```java
/*
 * Created on 14.05.2004
 */
package kiel.simulator.syncchart;

import java.util.ArrayList;
import java.util.Collection;
import java.util.HashSet;

import kiel.dataStructure.eventexp.IntegerSignal;
import kiel.dataStructure.eventexp.Signal;
import kiel.simulator.syncchart.exceptions.STRLException;
import kiel.simulator.syncchart.sigexp.STRLSignal;
import kiel.simulator.syncchart.sigexp.STRLSignalHandler;

/**
 * A SyncChart contains one STRLState and input/output signals.
 * @author Andre Ohlhoff
 */
public class SyncChart extends STRLReactingComponent {

    /**
     * signal handler for input signals.
     */
    private STRLSignalHandler input = new STRLSignalHandler();

    /**
     * signal handler for output signals.
     */
    private STRLSignalHandler output = new STRLSignalHandler();

    /**
     * the top state.
     */
    private STRLState top = null;

    /**
     * Sets the input signals.
     * @param in STRLSignals used as input
     */
    public final void setInput(final STRLSignal[] in) {
        this.input.setSignals(in);
    }

    /**
     * Sets the output signals.
     * @param out STRLSignals used as output
     */
    public final void setOutput(final STRLSignal[] out) {
        this.output.setSignals(out);
    }

    /**
     * Sets the top State.
     * @param t MacroState as syncchart root
     */
    public final void setTop(final STRLMacroState t) {
        this.top = t;
    }

    /**
     * Sets the context for the contained components.
     * @param sim STRLSimulator as context
     */
    public final void setComponentsContext(final STRLSimulator sim) {
        this.top.setContext(sim);
        this.input.setContext(sim);
        this.output.setContext(sim);
    }

    /**
     * Returns the list of active states, used to construct a Configuration.
     * @return Returns the list of active states, used to construct a
     *         Configuration.
     */
    public final ArrayList getActiveStates() {
        return this.top.getActiveStates();
    }

    /**
     * Starts the reaction.
     * @return Returns PAUSE, DEAD or UNKNOWN
     * @throws STRLException The exception is thrown if something went wrong.
     */
    public final int react() throws STRLException {
        return this.top.react();
    }

    /**
     * Prepares the SyncChart for the next instant.
     */
    public final void prepareForNext() {
        this.input.resetStatus();
        this.output.resetStatus();
        this.top.prepareForNext();
    }

    /**
     * Emits the corresponding STRLSignal.
     * @param signal kiel signal
     * @throws STRLException The exception is thrown if something went wrong.
     */
    public final void emit(final Signal signal) throws STRLException {
        this.input.emit(signal);
    }

    /**
     * Emits the corresponding STRLSignal with the given value.
     * @param signal kiel IntegerSignal
     * @param i value
     * @throws STRLException The exception is thrown if something went wrong.
     */
    public final void emit(final IntegerSignal signal, final int i)
            throws STRLException {
        this.input.emit(signal, i);

    }

    /**
     * Initializes the input and output signals. Must be called before the start
     * of the simulation.
     * @throws STRLException The exception is thrown if something went wrong.
     */
    public final void initInputAndOutput() throws STRLException {
        this.input.initialize();
        this.output.initialize();
    }

    /**
     * Returns the remaining potential.
     * @param fresh fresh signals
     * @param unknown unknown signals
     * @return Returns the remaining potential.
     */
    public final Collection getRemainingPotential(final Collection fresh,
            final Collection unknown) {
        Collection result = this.top.getRemainingPotential(fresh, unknown);
        this.resetHelpingFlags();
        return result;
    }

    /**
     * Updates the status of all signals, which are visible.
     * @param potential set of potentially emitted signals.
     * @see STRLReactingComponent#update(Collection)
     */
    public final void update(final Collection potential) {
        this.input.update(potential);
        this.output.update(potential);
        this.top.update(potential);
    }

    /**
     * At the end of a MacroStep this method updates all signals.
     */
    public final void finalUpdate() {
        this.input.update(new HashSet());
        this.output.update(new HashSet());
        this.top.finalUpdate();
    }

    /**
     * nothing happens.
     * @param thc is ignored
     */
    public final void deepEnter(final boolean thc) {
    }

    /**
     * Nothing happens.
     */
    public final void deepExit() {
    }

    /**
     * nothing happens.
     * @param fresh is ignored
     * @param unknown is ignored
     * @return Returns an empty set.
     */
    public final Collection getCompletePotential(final Collection fresh,
            final Collection unknown) {
        return new HashSet();
    }

    /**
     * Returns false.
     * @return Returns false.
     */
    public final boolean reachesFinal() {
        return false;
    }

    /**
     * Returns an empty set.
     * @return Return an empty set.
     */
    public final Collection getExitPotential() {
        return new HashSet();
    }

    /**
     * Resets some helping flags.
     */
    public final void resetHelpingFlags() {
        this.top.resetHelpingFlags();
    }

    /**
     * Activate the syncchart.
     */
    public final void activate() {
        this.top.activate();
    }

}
```

.4 kiel.simulator.syncchart.action

.4.1 STRLAction.java

```
/*
 * Created on 14.06.2004
 */
package kiel.simulator.syncchart.action;

import java.util.Collection;

import kiel.simulator.syncchart.STRLComponent;
import kiel.simulator.syncchart.exceptions.STRLException;
import kiel.simulator.syncchart.intexp.STRLIntegerVariable;

/**
 * Interface for all kinds of actions.
 * @author Andre Ohlhoff
 */
public abstract class STRLAction extends STRLComponent {

    /**
     * Executes the action.
     * @throws STRLException
     *              The exception is thrown if something went wrong like an
     *              uninitialized read of a variable.
     */
    public abstract void execute() throws STRLException;

    /**
     * Returns whether this action is ready or not.
     * @return Returns whether this action is ready or not.
     */
    public abstract boolean ready();

    /**
     * Returns a set of the emitted signal.
     * @return Returns a set of the emitted signal.
     */
    public abstract Collection getPotential();

    /**
     * Returns the collection of variables, which are read from the action.
     * @return Returns the collection of variables, which are read from the
     *         action.
     */
    public abstract Collection getVariablesRead();

    /**
     * Returns the variable, which is written or null if none is written.
     * @return Returns the variable, which is written or null if none is
     *         written.
     */
    public abstract STRLIntegerVariable getVariableWritten();

    /**
     * This method is used to do something before the 'real' effect.
     * @throws STRLException
     *              The exception is thrown if something went wrong.
     */
    public void prepare() throws STRLException {
    }
}
```

.4.2 STRLActions.java

```java
/*
 * Created on 14.05.2004
 */
package kiel.simulator.syncchart.action;

import java.util.Collection;
import java.util.HashSet;
import java.util.Iterator;
import java.util.LinkedList;

import kiel.simulator.syncchart.STRLComponent;
import kiel.simulator.syncchart.STRLSimulator;
import kiel.simulator.syncchart.exceptions.STRLException;

/**
 * Actions contains a list of action objects. Actions is used as a seperate
 * layer between action and transitions or states.
 * @author Andre Ohlhoff
 */
public class STRLActions extends STRLComponent {

    /**
     * List of actions.
     */
    private LinkedList actions = new LinkedList();

    /**
     * Constructs new empty actions.
     */
    public STRLActions() {
        this(new STRLAction[0]);
    }

    /**
     * Constructs new actions.
     * @param a
     *            Array of action to execute.
     */
    public STRLActions(final STRLAction[] a) {
        for (int i = 0; i < a.length; i++) {
            this.actions.add(a[i]);
        }
    }

    /**
     * Executes the single actions one after another.
     * @throws STRLException
     *             An Exception is thrown if a action is executed illegally.
     */
    public final void execute() throws STRLException {
        Iterator iter = this.actions.iterator();
        while (iter.hasNext()) {
            this.getContext().tryExecute((STRLAction) iter.next());
        }
    }

    /**
     * Returns the set of signals, which can be emitted by these actions.
     * @return Returns the set of signals, which can be emitted by these
     *         actions.
     */
    public final Collection getPotential() {
        HashSet result = new HashSet();
        Iterator iter = this.actions.iterator();
        while (iter.hasNext()) {
            result.addAll(((STRLAction) iter.next()).getPotential());
        }
        return result;
    }

    /**
     * Sets the context for the contained actions.
     * @param sim
     *            Context to be set.
     */
    public final void setComponentsContext(final STRLSimulator sim) {
        STRLComponent.setContext(sim, this.actions);
    }

    /**
     * Returns true, if the actions are empty.
     * @return Returns true, if the actions are empty.
     */
    public final boolean isEmpty() {
        return this.actions.isEmpty();
    }
}
```

.4.3 STRLAssignment.java

```
/*
 * Created on 14.06.2004
 */
package kiel.simulator.syncchart.action;

import java.util.Collection;
import java.util.HashSet;

import kiel.simulator.syncchart.exceptions.STRLUninitializedReadException;
import kiel.simulator.syncchart.intexp.STRLIntegerExpression;
import kiel.simulator.syncchart.intexp.STRLIntegerVariable;

/**
 * Assignment assigns the value of an expression to a variable if executed.
 * @author Andre Ohlhoff
 */
public class STRLAssignment extends STRLAction {

    /**
     * Variable to assign a value to.
     */
    private STRLIntegerVariable variable = null;

    /**
     * Expression to evaluate.
     */
    private STRLIntegerExpression intExp = null;

    /**
     * Constructs an assignment.
     * @param var
     *            variable
     * @param iExp
     *            expression to evaluate
     */
    public STRLAssignment(final STRLIntegerVariable var,
            final STRLIntegerExpression iExp) {
        this.variable = var;
        this.intExp = iExp;
    }

    /**
     * Assigns the value of the expression to the variable. The assignment can
     * only be executed, if all signals in the expression are ready.
     * @throws STRLUninitializedReadException
     *             The exception is thrown if a signal or variable in the
     *             expression has no valid value.
     */
    public final void execute() throws STRLUninitializedReadException {
        int value = this.intExp.getValue();
        this.variable.setValue(value);
    }

    /**
     * Returns an empty collection.
     * @return Returns an empty collection.
     */
    public final Collection getPotential() {
        return new HashSet();
    }

    /**
     * Returns whether the integer expression is ready.
     * @return Returns whether the integer expression is ready.
     */
    public final boolean ready() {
        return this.intExp.ready();
    }

    /**
     * Returns a collection of variables, which are used in the integer
     * expression.
     * @return Returns a collection of variables, which are used in the integer
     *         expression.
     */
    public final Collection getVariablesRead() {
        return this.intExp.getVariablesUsed();
    }

    /**
     * Returns the variable.
     * @return Returns the variable.
     */
    public final STRLIntegerVariable getVariableWritten() {
        return this.variable;
    }
}
```

.4.4 STRLDelayedAction.java

```
/*
 * Created on 05.11.2004
 */
package kiel.simulator.syncchart.action;

import java.util.Collection;
import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;

import kiel.simulator.syncchart.exceptions.STRLException;
import kiel.simulator.syncchart.intexp.STRLIntegerVariable;

/**
 * If an action is executed it becomes a delayed action if it depends on a not
 * yet ready signal or on the execution of another already delayed action.
 * @author Andre Ohlhoff
 */
public class STRLDelayedAction {

    /**
     * is true, if this delayed action was executed.
     */
    private boolean done = false;

    /**
     * the action to execute.
     */
    private STRLAction action = null;

    /**
     * the set of delayed action, which must be executed first.
     */
    private Set depends = new HashSet();

    /**
     * Constructs a new delayed action.
     * @param a action to delay possibly
     * @param other all delayed action; used to compute dependencies
     * @throws STRLException The exception is thrown if something went wrong.
     */
    public STRLDelayedAction(final STRLAction a, final Collection other)
            throws STRLException {
        this.action = a;

        this.action.prepare();

        Collection variablesRead = new HashSet();
        variablesRead.addAll(a.getVariablesRead());
        if (a.getVariableWritten() != null) {
            variablesRead.add(a.getVariableWritten());
        }
        if (!variablesRead.isEmpty()) {
            Iterator iter = other.iterator();
            while (iter.hasNext()) {
                STRLDelayedAction delayedAction = (STRLDelayedAction) iter
                    .next();
            STRLIntegerVariable var = delayedAction.getVariable();
            if (var != null && variablesRead.contains(var)) {
                this.depends.add(delayedAction);
            }
            if (a.getVariableWritten() != null
                    && delayedAction.getUsedVariables().contains(
                            a.getVariableWritten())) {
                this.depends.add(delayedAction);
            }
        }
    }
}

/**
 * Returns true, when the action was executed.
 * @return Returns true, when the action was executed.
 */
public final boolean isDone() {
    return this.done;
}

/**
 * Returns the variable, to which a value is assigned or null.
 * @return Returns the variable, to which a value is assigned or null.
 */
public final STRLIntegerVariable getVariable() {
    return this.action.getVariableWritten();
}

/**
 * Returns the collection of variables, which are used in the action.
 * @return Returns the collection of variables, which are used in the
 *         action.
 */
public final Collection getUsedVariables() {
    return this.action.getVariablesRead();
}

/**
 * Returns true, if the delayed action can be executed now.
 * @return Returns true, if the delayed action can be executed now.
 */
public final boolean ready() {
    Iterator iter = this.depends.iterator();
    while (iter.hasNext()) {
        if (!((STRLDelayedAction) iter.next()).isDone()) {
            return false;
        }
    }
    // all used vars are ready
    this.depends.clear();
    // check if signals are ready
    return this.action.ready();
}
```

```
    /**
     * Executes the action.
     * @throws STRLException The exception is thrown, if something went wrong.
     */
    public final void execute() throws STRLException {
        this.action.execute();
        this.done = true;
    }

    /**
     * Returns the potential of the action.
     * @return Returns the potential of the action.
     */
    public final Collection getPotential() {
        return this.action.getPotential();
    }
}
```

.4.5 STRLDelayedActionHandler.java

```
/*
 * Created on 22.10.2004
 */
package kiel.simulator.syncchart.action;

import java.util.Collection;
import java.util.HashSet;
import java.util.Iterator;

import kiel.simulator.syncchart.STRLComponent;
import kiel.simulator.syncchart.boolexp.STRLBooleanExpression;
import kiel.simulator.syncchart.exceptions.STRLException;

/**
 * The ActionHandler organizes the assinging of values to signals and variables.
 * @author Andre Ohlhoff
 */
public class STRLDelayedActionHandler extends STRLComponent {

    /**
     * Collection of all delayed actions.
     */
    private Collection delayedActions = new HashSet();

    /**
     * Tries to execute the action. If some not ready yet signal is used, or if
     * a dependency to a variable is revealed, the action becomes a delayed
     * action.
     * @param action
     *            action to execute now, or delay
     * @throws STRLException
     *             The exception is thrown is something went wrong.
     */
    public final void tryExecute(final STRLAction action) throws STRLException {
        STRLDelayedAction delayedAction = new STRLDelayedAction(action,
                delayedActions);
        if (delayedAction.ready()) {
            delayedAction.execute();
        } else {
            this.delayedActions.add(delayedAction);
        }
    }

    /**
     * Tries to execute the delayed actions.
     * @return Returns true if nothing is left to execute.
     * @throws STRLException
     *             The exception is thrown if something went wrong.
     */
    public final boolean execute() throws STRLException {
        boolean didSomething;
        do {
            didSomething = false;
            Collection done = new HashSet();
            Iterator iter = this.delayedActions.iterator();
            while (iter.hasNext()) {
                STRLDelayedAction action = (STRLDelayedAction) iter.next();
                if (action.ready()) {
                    action.execute();
                    done.add(action);
                    didSomething = true;
                }
            }
            this.delayedActions.removeAll(done);
            if (this.delayedActions.isEmpty()) {
                break;
            }
        } while (didSomething);
        return this.delayedActions.isEmpty();
    }

    /**
     * Returns the union of the potential of all delayed actions.
     * @return Returns the union of the potential of all delayed actions.
     */
    public final Collection getCompletePotential() {
        HashSet result = new HashSet();
        Iterator iter = this.delayedActions.iterator();
        while (iter.hasNext()) {
            result.addAll(((STRLDelayedAction) iter.next()).getPotential());
        }
        return result;
    }

    /**
     * Evaluates a guard. This method is used because the guard might contain
     * variables with delayed assignments.
     * @param guard
     *            guard to evaluate
     * @param fresh
     *            fresh signals
     * @param unknown
     *            unknown signals
     * @return TRUE, FALSE or UNKNOWNs
     */
    public final int evaluateGuard(final STRLBooleanExpression guard,
            final Collection fresh, final Collection unknown) {
        Collection usedVars = guard.getVariablesUsed();
        Iterator iter = this.delayedActions.iterator();
        while (iter.hasNext()) {
            STRLDelayedAction action = (STRLDelayedAction) iter.next();
            if (action.getVariable() != null
                    && usedVars.contains(action.getVariable())) {
                return STRLBooleanExpression.UNKNOWN;
            }
        }
        return guard.evaluate(fresh, unknown);
    }
}
```

.4.6 STRLEmitSignal.java

```
/*
 * Created on 14.06.2004
 */
package kiel.simulator.syncchart.action;

import java.util.Collection;
import java.util.HashSet;

import kiel.simulator.syncchart.intexp.STRLIntegerVariable;
import kiel.simulator.syncchart.sigexp.STRLPureSignal;

/**
 * EmitSignals contains a pure signal which is emitted.
 */
public class STRLEmitSignal extends STRLAction {

    /**
     * signal to emit.
     */
    private STRLPureSignal signal = null;

    /**
     * Constructs a EmitSignal.
     * @param s signal to emit.
     */
    public STRLEmitSignal(final STRLPureSignal s) {
        this.signal = s;
    }

    /**
     * Emits the signal.
     */
    public final void execute() {
        this.signal.emit();
    }

    /**
     * Returns a set containing only this signal.
     * @return Returns a set containing only this signal.
     */
    public final Collection getPotential() {
        HashSet result = new HashSet();
        result.add(signal);
        return result;
    }

    /**
     * Returns true.
     * @return Returns true.
     */
    public final boolean ready() {
        return true;
    }

    /**
     * Returns an empty collection.
     * @return Returns an empty collection.
     */
    public final Collection getVariablesRead() {
        return new HashSet();
    }

    /**
     * Returns null.
     * @return Returns null.
     */
    public final STRLIntegerVariable getVariableWritten() {
        return null;
    }
}
```

.4.7 STRLEmitValuedSignal.java

```
/*
 * Created on 14.06.2004
 */
package kiel.simulator.syncchart.action;

import java.util.Collection;
import java.util.HashSet;

import kiel.simulator.syncchart.exceptions.STRLException;
import kiel.simulator.syncchart.intexp.STRLIntegerExpression;
import kiel.simulator.syncchart.intexp.STRLIntegerVariable;
import kiel.simulator.syncchart.sigexp.STRLIntegerSignal;

/**
 * EmitValuedSignal emits a valued signal with a value of an expression.
 * @author Andre Ohlhoff
 */
public class STRLEmitValuedSignal extends STRLAction {

    /**
     * signal to emit.
     */
    private STRLIntegerSignal signal = null;

    /**
     * the value of the expression is used to emit the signal.
     */
    private STRLIntegerExpression intExp = null;

    /**
     * Constructs an EmitValuedSignal.
     * @param s signal
     * @param iExp integer expression
     */
    public STRLEmitValuedSignal(final STRLIntegerSignal s,
            final STRLIntegerExpression iExp) {
        this.signal = s;
        this.intExp = iExp;
    }

    /**
     * Emits the value of the signal.
     * @throws STRLException The exception is thrown if something went wrong.
     */
    public final void execute() throws STRLException {
        this.signal.emitValue(this.intExp.getValue());
    }

    /**
     * Emits the status of the signal.
     * @throws STRLException The exception is thrown if something went wrong.
     */
    public final void prepare() throws STRLException {
        this.signal.emit();
    }

    /**
     * Returns a collection only containing this signal.
     * @return Returns a collection only containing this signal.
     */
    public final Collection getPotential() {
        HashSet result = new HashSet();
        result.add(signal);
        return result;
    }

    /**
     * Returns whether the integer expression is ready.
     * @return Returns whether the integer expression is ready.
     */
    public final boolean ready() {
        return this.intExp.ready();
    }

    /**
     * Returns a collection of variables used in the integer expression.
     * @return Returns a collection of variables used in the integer
     * expression.
     */
    public final Collection getVariablesRead() {
        return this.intExp.getVariablesUsed();
    }

    /**
     * Returns null.
     * @return Returns null.
     */
    public final STRLIntegerVariable getVariableWritten() {
        return null;
    }
}
```

.4.8 STRLInitializeSignal.java

```
/*
 * Created on 05.11.2004
 */
package kiel.simulator.syncchart.action;

import java.util.Collection;
import java.util.HashSet;

import kiel.simulator.syncchart.exceptions.STRLException;
import kiel.simulator.syncchart.intexp.STRLIntegerExpression;
import kiel.simulator.syncchart.intexp.STRLIntegerVariable;
import kiel.simulator.syncchart.sigexp.STRLIntegerSignal;
import kiel.simulator.syncchart.sigexp.STRLSignal;

/**
 * @author Andre Ohlhoff
 */
public class STRLInitializeSignal extends STRLAction {

    /**
     * signal to initialize.
     */
    private STRLSignal signal = null;

    /**
     * Constructs a InitializeSignal action.
     * @param s
     *            signal to initialize
     */
    public STRLInitializeSignal(final STRLSignal s) {
        this.signal = s;
    }

    /**
     * Initializes the signal.
     * @throws STRLException
     *             The exception is thrown if something went wrong.
     */
    public final void execute() throws STRLException {
        this.signal.initializeValue();
        this.getContext().somethingChanged();
    }

    /**
     * Initializes the signal status.
     */
    public final void prepare() {
        this.getContext().debug(
                "try␣to␣init␣signal␣" + this.signal.getKielEvent().getName());
        this.signal.initializeStatus();
    }

    /**
     * Returns whether the signal can be initialized or not.
     * @return Returns whether the signal can be initialized or not.
     */
    public final boolean ready() {
        if (this.signal instanceof STRLIntegerSignal) {
            STRLIntegerSignal s = (STRLIntegerSignal) this.signal;
            if (s.mustBeInitialized()) {
                return s.getInitExpression().ready();
            }
        }
        return true;
    }

    /**
     * Returns a collection containing the signal.
     * @return Returns a collection containing the signal.
     */
    public final Collection getPotential() {
        HashSet result = new HashSet();
        result.add(this.signal);
        return result;
    }

    /**
     * Returns the collection of variables used to initialize this signal.
     * @return Returns the collection of variables used to initialize this
     *         signal.
     */
    public final Collection getVariablesRead() {
        if (this.signal instanceof STRLIntegerSignal) {
            STRLIntegerSignal s = (STRLIntegerSignal) this.signal;
            if (s.mustBeInitialized()) {
                STRLIntegerExpression exp = s.getInitExpression();
                return exp.getVariablesUsed();
            }
        }
        return new HashSet();
    }

    /**
     * Returns null.
     * @return Returns null.
     */
    public final STRLIntegerVariable getVariableWritten() {
        return null;
    }
}
```

.4.9 STRLInitializeTransition.java

```java
/*
 * Created on 06.11.2004
 */
package kiel.simulator.syncchart.action;

import java.util.Collection;
import java.util.HashSet;

import kiel.simulator.syncchart.exceptions.STRLException;
import kiel.simulator.syncchart.intexp.STRLIntegerVariable;
import kiel.simulator.syncchart.transition.STRLTransition;

/**
 * InitializeTransition is used to initialize a transition.
 * @author Andre Ohlhoff
 */
public class STRLInitializeTransition extends STRLAction {

    /**
     * transition to initialize.
     */
    private STRLTransition transition = null;

    /**
     * Constructs a InitializeTransition.
     * @param t transition to initialize
     */
    public STRLInitializeTransition(final STRLTransition t) {
        this.transition = t;
    }

    /**
     * Initializes the transition.
     * @throws STRLException The exception is thrown if something went wrong
     */
    public final void execute() throws STRLException {
        this.transition.init();
    }

    /**
     * Returns true, if the transition can be initialized.
     * @return Returns true, if the transition can be initialized.
     */
    public final boolean ready() {
        return this.transition.getInitExpression().ready();
    }

    /**
     * Returns an empty set.
     * @return Returns an empty set.
     */
    public final Collection getPotential() {
        return new HashSet();
    }

    /**
     * Return the collection of all used variables.
     * @return Return the collection of all used variables.
     */
    public final Collection getVariablesRead() {
        return this.transition.getInitExpression().getVariablesUsed();
    }

    /**
     * Returns null.
     * @return Returns null.
     */
    public final STRLIntegerVariable getVariableWritten() {
        return null;
    }

    /**
     * Sets the transitions isInitialized flag to false.
     */
    public final void prepare() {
        this.getContext().debug(
                "try␣to␣init␣transition:␣"
                    + this.transition.getKielTransition().toString());
        this.transition.setUnitialized();
    }
}
```

.4.10 STRLInitializeVariable.java

```java
/*
 * Created on 05.11.2004
 */
package kiel.simulator.syncchart.action;

import java.util.Collection;
import java.util.HashSet;

import kiel.simulator.syncchart.exceptions.STRLException;
import kiel.simulator.syncchart.intexp.STRLIntegerExpression;
import kiel.simulator.syncchart.intexp.STRLIntegerVariable;

/**
 * @author Andre Ohlhoff
 */
public class STRLInitializeVariable extends STRLAction {

    /**
     * variable to initialize.
     */
    private STRLIntegerVariable variable = null;

    /**
     * Constructs a InitializeVariable action.
     * @param v
     *            variable to initialize
     */
    public STRLInitializeVariable(final STRLIntegerVariable v) {
        this.variable = v;
    }

    /**
     * Initializes the variable.
     * @throws STRLException
     *             The exception is thrown if something went wrong.
     */
    public final void execute() throws STRLException {
        this.variable.initializeValue();
    }

    /**
     * Returns whether the variable can be initialized or not.
     * @return Returns whether the variable can be initialized or not.
     */
    public final boolean ready() {
        if (this.variable.mustBeInitialized()) {
            return this.variable.getInitExpression().ready();
        }
        return true;
    }

    /**
     * Returns an empty set.
     * @return Returns an empty set.
     */
    public final Collection getPotential() {
        HashSet result = new HashSet();
        return result;
    }

    /**
     * Returns the collection of variables used to initialize this variable.
     * @return Returns the collection of variables used to initialize this
     *         variable.
     */
    public final Collection getVariablesRead() {
        if (this.variable.mustBeInitialized()) {
            STRLIntegerExpression exp = this.variable.getInitExpression();
            return exp.getVariablesUsed();
        }
        return new HashSet();
    }

    /**
     * Returns the variable.
     * @return Returns the variable.
     */
    public final STRLIntegerVariable getVariableWritten() {
        return this.variable;
    }

    /**
     * prints a debug message.
     */
    public final void prepare() {
        this.getContext().debug(
                "try␣to␣initialize␣variable␣"
                        + this.variable.getKielVar().getName());
    }
}
```

.5 kiel.simulator.syncchart.boolexp

.5.1 STRLAnd.java

```
/*
 * Created on 14.06.2004
 *
 */
package kiel.simulator.syncchart.boolexp;

import java.util.Collection;
import java.util.HashSet;

/**
 * An And consists of two boolean expressions.
 * @author Andre Ohlhoff
 */
public class STRLAnd implements STRLBooleanExpression {

    /**
     * Returns the value of 'a AND b'.
     * @param a left side
     * @param b right side
     * @return Returns the value of 'a AND b'.
     */
    public static int eval(final int a, final int b) {

        if (a == STRLBooleanExpression.FALSE
                || b == STRLBooleanExpression.FALSE) {
            // one part is false
            return STRLBooleanExpression.FALSE;
        } else if (a == STRLBooleanExpression.UNKNOWN
                || b == STRLBooleanExpression.UNKNOWN) {
            // no part is false, at least one part is unknown
            return STRLBooleanExpression.UNKNOWN;
        } else {
            // both parts are true
            return STRLBooleanExpression.TRUE;
        }
    }

    /**
     * left side.
     */
    private STRLBooleanExpression left = null;

    /**
     * right side.
     */
    private STRLBooleanExpression right = null;

    /**
     * Constructs a new AND with the give left and rigt part.
     * @param l left part
     * @param r right part
     */
    public STRLAnd(final STRLBooleanExpression l,
            final STRLBooleanExpression r) {
        this.left = l;
        this.right = r;
    }

    /**
     * Returns the left part.
     * @return Returns the left part.
     */
    public final STRLBooleanExpression getLeft() {
        return this.left;
    }

    /**
     * Returns the right part.
     * @return Returns the right part.
     */
    public final STRLBooleanExpression getRight() {
        return this.right;
    }

    /**
     * Returns the value of the AND.
     * @param fresh fresh signals
     * @param unknown unknown signals
     * @return Returns the value of the AND.
     * @see STRLBooleanExpression#evaluate(Collection, Collection)
     */
    public final int evaluate(final Collection fresh,
            final Collection unknown) {
        return STRLAnd.eval(this.getLeft().evaluate(fresh, unknown), this
                .getRight().evaluate(fresh, unknown));

    }

    /**
     * Returns the value of the AND in the previous instant.
     * @param fresh fresh signals
     * @param unknown unknown signals
     * @return Returns the value of the AND in the previous instant.
     * @see STRLBooleanExpression#pre0(Collection, Collection)
     */
    public final int pre0(final Collection fresh, final Collection unknown) {
        return STRLAnd.eval(this.getLeft().pre0(fresh, unknown), this
                .getRight().pre0(fresh, unknown));
    }

    /**
     * Returns the value of the AND in the previous instant.
     * @param fresh fresh signals
     * @param unknown unknown signals
 * @return Returns the value of the AND in the previous instant.
 * @see STRLBooleanExpression#pre0(Collection, Collection)
 */
public final int pre1(final Collection fresh, final Collection unknown) {
    return STRLAnd.eval(this.getLeft().pre1(fresh, unknown), this
            .getRight().pre1(fresh, unknown));
}

/**
 * Returns the collection of all used variables.
 * @return Returns the collection of all used variables.
 */
public final Collection getVariablesUsed() {
    HashSet result = new HashSet();
    result.addAll(this.left.getVariablesUsed());
    result.addAll(this.right.getVariablesUsed());
    return result;
}
}
```

.5.2 STRLBooleanExpression.java

```
/*
 * Created on 14.06.2004
 */
package kiel.simulator.syncchart.boolexp;

import java.util.Collection;

/**
 * BooleanExpression is the interface for different kinds of boolean expressions
 * used in triggers or guards. A boolean expression only made of AND, OR, NOT,
 * PRE and signals is also called a signal expression. Boolean expression can
 * have three values: true, false and unknown.
 * @author Andre Ohlhoff
 */
public interface STRLBooleanExpression {

    /**
     * Constant used for 'false'.
     */
    int FALSE = 0;

    /**
     * Constant used for 'true'.
     */
    int TRUE = 1;

    /**
     * Constant used for 'unknown'.
     */
    int UNKNOWN = 2;

    /**
     * Evaluates the boolean expression. The parameters are ignored if the
     * boolean expression is not a signal expression.
     * @param fresh fresh signals
     * @param unknown unknown signals
     * @return Returns the value of the expression.
     */
    int evaluate(Collection fresh, final Collection unknown);

    /**
     * Returns the value of the expression in the previous instant. This is
     * only
     * defined for signal expressions. Signals with no previous state are
     * handled as if they were absent.
     * @param fresh fresh signals
     * @param unknown unknown signals.
     * @return Returns the value of the expression in the previous instant.
     */
    int pre0(Collection fresh, final Collection unknown);

    /**
     * Returns the value of the expression in the previous instant. This is only
     * defined for signal expressions. Signals with no previous state are
     * handled as if they were present.
     * @param fresh fresh signals
     * @param unknown unknown signals.
     * @return Returns the value of the expression in the previous instant.
     */
    int pre1(Collection fresh, final Collection unknown);

    /**
     * Returns the collection of all used variables.
     * @return Returns the collection of all used variables.
     */
    Collection getVariablesUsed();
}
```

.5.3 STRLBrackets.java

```
/*
 * Created on 15.09.2004
 */
package kiel.simulator.syncchart.boolexp;

import java.util.Collection;

/**
 * This is a bracketed boolean expression.
 * @author Andre Ohlhoff
 */
public class STRLBrackets implements STRLBooleanExpression {

    /**
     * Expression between brackets.
     */
    private STRLBooleanExpression body = null;

    /**
     * Constructs a bracketed boolean expression.
     * @param b Expression to be brackteted
     */
    public STRLBrackets(final STRLBooleanExpression b) {
        this.body = b;
    }

    /**
     * Evaluates the expression.
     * @param fresh fresh signals
     * @param unknown unknown signals
     * @return Returns the value of the expression
     * @see STRLBooleanExpression#evaluate(Collection, Collection)
     */
    public final int evaluate(final Collection fresh,
            final Collection unknown) {
        return this.body.evaluate(fresh, unknown);
    }

    /**
     * Returns the value of the expression in the previous instant.
     * @param fresh fresh signals
     * @param unknown unknown signals
     * @return Returns the value of the expression in the previous instant.
     * @see STRLBooleanExpression#pre0(Collection, Collection)
     */
    public final int pre0(final Collection fresh, final Collection unknown) {
        return this.body.pre0(fresh, unknown);
    }

    /**
     * Returns the value of the expression in the previous instant.
     * @param fresh fresh signals
     * @param unknown unknown signals
     * @return Returns the value of the expression in the previous instant.
     * @see STRLBooleanExpression#pre1(Collection, Collection)
     */
    public final int pre1(final Collection fresh, final Collection unknown) {
        return this.body.pre1(fresh, unknown);
    }

    /**
     * Returns the collection of all used variables.
     * @return Returns the collection of all used variables.
     */
    public final Collection getVariablesUsed() {
        return this.body.getVariablesUsed();
    }

}
```

.5.4 STRLComparator.java

```java
/*
 * Created on 14.06.2004
 */
package kiel.simulator.syncchart.boolexp;

import java.util.Collection;
import java.util.HashSet;

import kiel.simulator.syncchart.exceptions.STRLUninitializedReadException;
import kiel.simulator.syncchart.intexp.STRLIntegerExpression;

/**
 * Comparator is the basis class for all used integer relations.
 * @author Andre Ohlhoff
 */
public abstract class STRLComparator implements STRLBooleanExpression {

    /**
     * Left integer expression.
     */
    private STRLIntegerExpression left = null;

    /**
     * Right integer expression.
     */
    private STRLIntegerExpression right = null;

    /**
     * Constructs a new Comparator with a left and a right side.
     * @param l left side
     * @param r right side
     */
    public STRLComparator(final STRLIntegerExpression l,
            final STRLIntegerExpression r) {
        this.left = l;
        this.right = r;
    }

    /**
     * Returns the result of the relation.
     * @param fresh fresh signals
     * @param unknown unknown signals
     * @return Returns the result of the relation.
     * @see STRLBooleanExpression#evaluate(Collection, Collection)
     */
    public final int evaluate(final Collection fresh,
            final Collection unknown) {
        if (this.left.ready() && this.right.ready()) {
            try {
                int l = this.left.getValue();
                int r = this.right.getValue();
                if (this.compare(l, r)) {
                    return STRLBooleanExpression.TRUE;
                } else {
                    return STRLBooleanExpression.FALSE;
                }
            } catch (STRLUninitializedReadException e) {
                // dead code
                System.out.println("ohoh");
            }
            // dead code
            return -1;

        } else {
            return STRLBooleanExpression.UNKNOWN;
        }

    }

    /**
     * Comparing method which must be overridden in extending classes.
     * @param a left integer value
     * @param b right integer value
     * @return Returns the result of the relation.
     */
    public abstract boolean compare(final int a, final int b);

    /**
     * This method is only used for signal expressions.
     * @param fresh ignored
     * @param unknown ignored
     * @return nothing
     * @see STRLBooleanExpression#pre0(Collection, Collection)
     */
    public final int pre0(final Collection fresh,
            final Collection unknown) {
        return -1; // dead code
    }

    /**
     * This method is only used for signal expressions.
     * @param fresh ignored
     * @param unknown ignored
     * @return nothing
     * @see STRLBooleanExpression#pre1(Collection, Collection)
     */
    public final int pre1(final Collection fresh,
            final Collection unknown) {
        return -1; // dead code
    }

    /**
     * Returns the collection of all used variables.
     * @return Returns the collection of all used variables.
     */
    public final Collection getVariablesUsed() {
        HashSet result = new HashSet();
        result.addAll(this.left.getVariablesUsed());
        result.addAll(this.right.getVariablesUsed());
        return result;
    }
}
```

.5.5 STRLEqual.java

```
/*
 * Created on 14.06.2004
 */
package kiel.simulator.syncchart.boolexp;

import kiel.simulator.syncchart.intexp.STRLIntegerExpression;

/**
 * The == relation.
 * @author Andre Ohlhoff
 */
public class STRLEqual extends STRLComparator {

    /**
     * Constructs a == relation with the given left and right integer
     * expression.
     * @param left left expression
     * @param right right expression
     */
    public STRLEqual(final STRLIntegerExpression left,
            final STRLIntegerExpression right) {
        super(left, right);
    }

    /**
     * Compares two values with ==.
     * @param a left value
     * @param b right value
     * @return Returns (a == b)
     */
    public final boolean compare(final int a, final int b) {
        return a == b;
    }
}
```

.5.6 STRLFalse.java

```
/*
 * Created on 14.06.2004
 */
package kiel.simulator.syncchart.boolexp;

import java.util.Collection;
import java.util.HashSet;

/**
 * False is always false.
 * @author Andre Ohlhoff
 */
public class STRLFalse implements STRLBooleanExpression {

    /**
     * Returns false.
     * @param fresh
     *            ignored
     * @param unknown
     *            ignored
     * @return Returns false.
     * @see STRLBooleanExpression#evaluate(Collection, Collection)
     */
    public final int evaluate(final Collection fresh,
            final Collection unknown) {
        return STRLBooleanExpression.FALSE;
    }

    /**
     * This method is only used for signal expressions.
     * @param fresh
     *            ignored
     * @param unknown
     *            ignored
     * @return nothing
     * @see STRLBooleanExpression#pre0(Collection, Collection)
     */
    public final int pre0(final Collection fresh, final Collection unknown) {
        return -1; // dead code
    }

    /**
     * This method is only used for signal expressions.
     * @param fresh
     *            ignored
     * @param unknown
     *            ignored
     * @return nothing
     * @see STRLBooleanExpression#pre1(Collection, Collection)
     */
    public final int pre1(final Collection fresh, final Collection unknown) {
        return -1; // dead code
    }

    /**
     * Returns the collection of all used variables.
     * @return Returns the collection of all used variables.
     */
    public final Collection getVariablesUsed() {
        return new HashSet();
    }

}
```

.5.7 STRLGreaterOrEqual.java

```
/*
 * Created on 14.06.2004
 */
package kiel.simulator.syncchart.boolexp;

import kiel.simulator.syncchart.intexp.STRLIntegerExpression;

/**
 * The &gt;= relation.
 * @author Andre Ohlhoff
 */
public class STRLGreaterOrEqual extends STRLComparator {

    /**
     * * Constructs a &gt;= relation with the given left and right integer
     * expression.
     * @param left left expression
     * @param right right expression
     */
    public STRLGreaterOrEqual(final STRLIntegerExpression left,
            final STRLIntegerExpression right) {
        super(left, right);
    }

    /**
     * Compares two values with &gt;=.
     * @param a left value
     * @param b right value
     * @return Returns (a &gt;= b)
     */
    public final boolean compare(final int a, final int b) {
        return a >= b;
    }

}
```

.5.8 STRLGreaterThan.java

```java
/*
 * Created on 14.06.2004
 */
package kiel.simulator.syncchart.boolexp;

import kiel.simulator.syncchart.intexp.STRLIntegerExpression;

/**
 * The &gt; relation.
 * @author Andre Ohlhoff
 */
public class STRLGreaterThan extends STRLComparator {

    /**
     * * Constructs a &gt; relation with the given left and right integer
     * expression.
     * @param left left expression
     * @param right right expression
     */
    public STRLGreaterThan(final STRLIntegerExpression left,
            final STRLIntegerExpression right) {
        super(left, right);
    }

    /**
     * Compares two values with &gt;.
     * @param a left value
     * @param b right value
     * @return Returns (a &gt; b)
     */
    public final boolean compare(final int a, final int b) {
        return a > b;
    }

}
```

.5.9 STRLLessOrEqual.java

```
/*
 * Created on 14.06.2004
 */
package kiel.simulator.syncchart.boolexp;

import kiel.simulator.syncchart.intexp.STRLIntegerExpression;

/**
 * The &lt;= relation.
 * @author Andre Ohlhoff
 */
public class STRLLessOrEqual extends STRLComparator {

    /**
     * * Constructs a &lt;= relation with the given left and right integer
     * expression.
     * @param left left expression
     * @param right right expression
     */
    public STRLLessOrEqual(final STRLIntegerExpression left,
            final STRLIntegerExpression right) {
        super(left, right);
    }

    /**
     * Compares two values with &lt;=.
     * @param a left value
     * @param b right value
     * @return Returns (a &lt;= b)
     */
    public final boolean compare(final int a, final int b) {
        return a <= b;
    }

}
```

.5.10 STRLLessThan.java

```java
/*
 * Created on 14.06.2004
 */
package kiel.simulator.syncchart.boolexp;

import kiel.simulator.syncchart.intexp.STRLIntegerExpression;

/**
 * The &lt; relation.
 * @author Andre Ohlhoff
 */
public class STRLLessThan extends STRLComparator {

    /**
     * Constructs a &lt; relation with the given left and right integer
     * expression.
     * @param left left expression
     * @param right right expression
     */
    public STRLLessThan(final STRLIntegerExpression left,
            final STRLIntegerExpression right) {
        super(left, right);
    }

    /**
     * Compares two values with &lt;.
     * @param a left value
     * @param b right value
     * @return Returns (a &lt; b)
     */
    public final boolean compare(final int a, final int b) {
        return a < b;
    }

}
```

.5.11 STRLNot.java

```java
/*
 * Created on 14.06.2004
 */
package kiel.simulator.syncchart.boolexp;

import java.util.Collection;

/**
 * The Negation of a boolean expression.
 * @author Andre Ohlhoff
 */
public class STRLNot implements STRLBooleanExpression {

    /**
     * Negates the given value.
     * @param a
     *            value to negate
     * @return Returns the negation of the given value.
     */
    public static int eval(final int a) {
        if (a == STRLBooleanExpression.TRUE) {
            return STRLBooleanExpression.FALSE;
        } else if (a == STRLBooleanExpression.FALSE) {
            return STRLBooleanExpression.TRUE;
        } else {
            return STRLBooleanExpression.UNKNOWN;
        }
    }

    /**
     * Expression which is negated.
     */
    private STRLBooleanExpression body = null;

    /**
     * Constructs a new negation of the given expression.
     * @param b
     *            expression to be negated.
     */
    public STRLNot(final STRLBooleanExpression b) {
        this.body = b;
    }

    /**
     * Returns the negated value.
     * @param fresh
     *            fresh signals
     * @param unknown
     *            unknown signals
     * @return Returns the negated value.
     * @see STRLBooleanExpression#evaluate(Collection, Collection)
     */
    public final int evaluate(final Collection fresh,
            final Collection unknown) {
        int code = this.body.evaluate(fresh, unknown);
        return STRLNot.eval(code);
    }

    /**
     * Returns the value of the previous instant.
     * @param fresh
     *            fresh signals
     * @param unknown
     *            unknown signals
     * @return Returns the value of the previous instant.
     * @see STRLBooleanExpression#pre0(Collection, Collection)
     */
    public final int pre0(final Collection fresh, final Collection unknown) {
        return STRLNot.eval(this.body.pre1(fresh, unknown));
    }

    /**
     * Returns the value of the previous instant.
     * @param fresh
     *            fresh signals
     * @param unknown
     *            unknown signals
     * @return Returns the value of the previous instant.
     * @see STRLBooleanExpression#pre1(Collection, Collection)
     */
    public final int pre1(final Collection fresh, final Collection unknown) {
        return STRLNot.eval(this.body.pre0(fresh, unknown));
    }

    /**
     * Returns the collection of all used variables.
     * @return Returns the collection of all used variables.
     */
    public final Collection getVariablesUsed() {
        return this.body.getVariablesUsed();
    }

}
```

.5.12 STRLNotEqual.java

```
/*
 * Created on 14.06.2004
 */
package kiel.simulator.syncchart.boolexp;

import kiel.simulator.syncchart.intexp.STRLIntegerExpression;

/**
 * The != relation.
 * @author Andre Ohlhoff
 */
public class STRLNotEqual extends STRLComparator {

    /**
     * Constructs a != relation with the given left and right integer
     * expression.
     * @param left left expression
     * @param right right expression
     */
    public STRLNotEqual(final STRLIntegerExpression left,
            final STRLIntegerExpression right) {
        super(left, right);
    }

    /**
     * Compares two values with !=.
     * @param a left value
     * @param b right value
     * @return Returns (a != b)
     */
    public final boolean compare(final int a, final int b) {
        return a != b;
    }

}
```

.5.13 STRLOr.java

```
/*
 * Created on 14.06.2004
 */
package kiel.simulator.syncchart.boolexp;

import java.util.Collection;
import java.util.HashSet;

/**
 * An Or consists of two boolean expressions.
 * @author Andre Ohlhoff
 */
public class STRLOr implements STRLBooleanExpression {

    /**
     * Returns the value of 'a OR b'.
     * @param a left side
     * @param b right side
     * @return Returns the value of 'a OR b'.
     */
    public static int eval(final int a, final int b) {
        if (a == STRLBooleanExpression.TRUE
                || b == STRLBooleanExpression.TRUE) {
            // one side is true
            return STRLBooleanExpression.TRUE;
        } else if (a == STRLBooleanExpression.UNKNOWN
                || b == STRLBooleanExpression.UNKNOWN) {
            // no side is true and at least one is unknown
            return STRLBooleanExpression.UNKNOWN;
        } else {
            // no side is true and no side is unknown
            return STRLBooleanExpression.FALSE;
        }

    }

    /**
     * left side.
     */
    private STRLBooleanExpression left = null;

    /**
     * right side.
     */
    private STRLBooleanExpression right = null;

    /**
     * Constructs a new Or with the give left and rigt part.
     * @param l left part
     * @param r right part
     */
    public STRLOr(final STRLBooleanExpression l,
            final STRLBooleanExpression r) {
        this.left = l;
        this.right = r;
    }

    /**
     * Returns the left part.
     * @return Returns the left part.
     */
    public final STRLBooleanExpression getLeft() {
        return this.left;
    }

    /**
     * Returns the right part.
     * @return Returns the right part.
     */
    public final STRLBooleanExpression getRight() {
        return this.right;
    }

    /**
     * Returns the value of the Or.
     * @param fresh fresh signals
     * @param unknown unknown signals
     * @return Returns the value of the Or.
     * @see STRLBooleanExpression#evaluate(Collection, Collection)
     */
    public final int evaluate(final Collection fresh,
            final Collection unknown) {
        return STRLOr.eval(this.getLeft().evaluate(fresh, unknown), this
                .getRight().evaluate(fresh, unknown));
    }

    /**
     * Returns the value of the Or in the previous instant.
     * @param fresh fresh signals
     * @param unknown unknown signals
     * @return Returns the value of the Or in the previous instant.
     * @see STRLBooleanExpression#pre0(Collection, Collection)
     */
    public final int pre0(final Collection fresh, final Collection unknown) {
        return STRLOr.eval(this.getLeft().pre0(fresh, unknown), this.getRight()
                .pre0(fresh, unknown));
    }

    /**
     * Returns the value of the Or in the previous instant.
     * @param fresh fresh signals
     * @param unknown unknown signals
     * @return Returns the value of the Or in the previous instant.
     * @see STRLBooleanExpression#pre1(Collection, Collection)
     */
    public final int pre1(final Collection fresh, final Collection unknown) {
        return STRLOr.eval(this.getLeft().pre1(fresh, unknown), this.getRight()
                .pre1(fresh, unknown));
    }

    /**
     * Returns the collection of all used variables.
     * @return Returns the collection of all used variables.
     */
    public final Collection getVariablesUsed() {
        HashSet result = new HashSet();
        result.addAll(this.left.getVariablesUsed());
        result.addAll(this.right.getVariablesUsed());
        return result;
    }
}
```

.5.14 STRLTrue.java

```
/*
 * Created on 14.06.2004
 */
package kiel.simulator.syncchart.boolexp;

import java.util.Collection;
import java.util.HashSet;

/**
 * STRLTrue is always true.
 * @author Andre Ohlhoff
 */
public class STRLTrue implements STRLBooleanExpression {

    /**
     * Returns true.
     * @param fresh ignored
     * @param unknown ignored
     * @return Returns true.
     * @see STRLBooleanExpression#evaluate(Collection, Collection)
     */
    public final int evaluate(final Collection fresh,
            final Collection unknown) {
        return STRLBooleanExpression.TRUE;
    }

    /**
     * This method is only used for signal expressions.
     * @param fresh ignored
     * @param unknown ignored
     * @return nothing
     * @see STRLBooleanExpression#pre0(Collection, Collection)
     */
    public final int pre0(final Collection fresh, final Collection unknown) {
        return -1; // dead code
    }

    /**
     * This method is only used for signal expressions.
     * @param fresh ignored
     * @param unknown ignored
     * @return nothing
     * @see STRLBooleanExpression#pre1(Collection, Collection)
     */
    public final int pre1(final Collection fresh, final Collection unknown) {
        return -1; // dead code
    }

    /**
     * Returns the collection of all used variables.
     * @return Returns the collection of all used variables.
     */
    public final Collection getVariablesUsed() {
        return new HashSet();
    }

}
```

.6 kiel.simulator.syncchart.converter

.6.1 ActionConverter.java

```
/*
 * Created on 30.06.2004
 */
package kiel.simulator.syncchart.converter;

import kiel.dataStructure.action.GenerateEvent;
import kiel.dataStructure.action.GenerateValuedEvent;
import kiel.dataStructure.action.IntegerAssignment;
import kiel.dataStructure.action.Action;
import kiel.dataStructure.eventexp.Event;
import kiel.dataStructure.eventexp.IntegerSignal;
import kiel.dataStructure.eventexp.Signal;
import kiel.simulator.syncchart.action.STRLAssignment;
import kiel.simulator.syncchart.action.STRLEmitSignal;
import kiel.simulator.syncchart.action.STRLEmitValuedSignal;
import kiel.simulator.syncchart.action.STRLAction;
import kiel.simulator.syncchart.exceptions.STRLException;
import kiel.simulator.syncchart.exceptions.STRLNotSupportedException;
import kiel.simulator.syncchart.intexp.STRLIntegerExpression;
import kiel.simulator.syncchart.intexp.STRLIntegerVariable;
import kiel.simulator.syncchart.sigexp.STRLIntegerSignal;

/**
 * ActionConverter is used to convert kiel action to syncchart STRLAction.
 * @author Andre Ohlhoff
 */
public class ActionConverter extends Context {

    /**
     * Constructs an ActionConverter.
     * @param context
     *            context for debug messages and signal/variable searching.
     */
    public ActionConverter(final Context context) {
        super(context);
    }

    /**
     * Converts a kiel action into a syncchart STRLAction.
     * @param action
     *            kiel action
     * @return syncchart action
     * @throws STRLException
     *             The exception is thrown if something is not convertable.
     */
    public final STRLAction convert(final Action action) throws STRLException {
        if (action instanceof IntegerAssignment) {
            IntegerAssignment ia = (IntegerAssignment) action;
            STRLIntegerVariable v = this.toIntVariable(ia.getVariable());
            IntExpConverter iConv = new IntExpConverter(this);
            STRLIntegerExpression iExp = iConv.convert(ia.getIntExp());
            return new STRLAssignment(v, iExp);
        } else if (action instanceof GenerateValuedEvent) {
            GenerateValuedEvent gve = (GenerateValuedEvent) action;
            Event e = gve.getEvent();
            if (e instanceof IntegerSignal) {
                STRLIntegerSignal s = this.toIntegerSignal((IntegerSignal) e);
                IntExpConverter iConv = new IntExpConverter(this);
                STRLIntegerExpression iExp = iConv.convert(gve.getExp());
                return new STRLEmitValuedSignal(s, iExp);
            } else {
                throw new STRLNotSupportedException("cannot␣use␣" + e
                        + "in␣GenerateValuedEvent");
            }
        } else if (action instanceof GenerateEvent) {
            GenerateEvent ge = (GenerateEvent) action;
            Event e = ge.getEvent();
            if ((e instanceof Signal) && !(e instanceof IntegerSignal)) {
                return new STRLEmitSignal(this.toSignal((Signal) e));
            } else {
                throw new STRLNotSupportedException("cannot␣use␣" + e
                        + "in␣GenerateEvent");
            }
        } else {
            throw new STRLNotSupportedException("cannot␣use␣" + action
                    + "␣as␣action");
        }
    }
}
```

.6.2 ActionsConverter.java

```java
/*
 * Created on 30.06.2004
 */
package kiel.simulator.syncchart.converter;

import kiel.dataStructure.CompoundLabel;
import kiel.dataStructure.StringLabel;
import kiel.dataStructure.TransitionLabel;
import kiel.dataStructure.action.Actions;
import kiel.simulator.syncchart.action.STRLActions;
import kiel.simulator.syncchart.action.STRLAction;
import kiel.simulator.syncchart.exceptions.STRLException;

/**
 * An ActionsConverter is used to convert kiel actions into STRLActions.
 * @author Andre Ohlhoff
 */
public class ActionsConverter extends Context {

    /**
     * Constructs a ActionsConverter.
     * @param context context for debug messages and signal/variable searching.
     */
    public ActionsConverter(final Context context) {
        super(context);
    }

    /**
     * Generates syncchart actions out of a kiel transitionlabel.
     * @param label kiel transitionlabel
     * @return syncchart actions.
     * @throws STRLException The exception is thrown if something is not
     *             convertable.
     */
    public final STRLActions convert(final TransitionLabel label)
            throws STRLException {
        if (label instanceof CompoundLabel) {
            Actions a = ((CompoundLabel) label).getEffect();
            return this.convert(a);
        } else if (!((StringLabel) label).toString().equals("")) {
            this.warning("StringLabel:␣\"" + label.toString() + "\"␣ignored");
        }
        return new STRLActions();

    }

    /**
     * Converts kiel actions to STRLActions.
     * @param a kiel actions
     * @return STRLActions
     * @throws STRLException The exception is thrown if something is not
     *             convertable.
     */
    public final STRLActions convert(final Actions a) throws STRLException {
        STRLAction[] actions = new STRLAction[a.getActions().length];

        // this.debug("action: " + a.toString());
        ActionConverter aConv = new ActionConverter(this);

        for (int i = 0; i < actions.length; i++) {
            actions[i] = aConv.convert(a.getActions()[i]);
        }

        STRLActions result = new STRLActions(actions);
        return result;
    }

}
```

.6.3 ANDToMacro.java

```
/*
 * Created on 18.06.2004
 */
package kiel.simulator.syncchart.converter;

import java.util.Iterator;

import kiel.dataStructure.ANDState;
import kiel.dataStructure.FinalANDState;
import kiel.dataStructure.Region;
import kiel.simulator.syncchart.STRLMacroState;
import kiel.simulator.syncchart.STRLSTG;
import kiel.simulator.syncchart.exceptions.STRLException;
import kiel.simulator.syncchart.exceptions.STRLNotSupportedException;

/**
 * ANDToMacro converts a kiel ANDState to a STRLMacroState.
 * @author Andre Ohlhoff
 */
public class ANDToMacro extends Context {

    /**
     * Constructs a ANDToMacro converter.
     * @param context
     *            context for debug messages and signal/variable searching.
     */
    public ANDToMacro(final Context context) {
        super(context);
    }

    /**
     * Converts the kiel ANDState to a STRLMacroState.
     * @param state
     *            kiel ANDState
     * @return STRLMacroState
     * @throws STRLException
     *             The exception is thrown if something is not convertable.
     */
    public final STRLMacroState convert(final ANDState state)
            throws STRLException {
        this.debug("converting:␣" + state.getID());

        STRLMacroState result = new STRLMacroState(state);

        result.setLocalSignals(this.convertSignals(state.getLocalEvents()));

        Iterator regionIterator = state.getSubnodes().iterator();

        STRLSTG[] stgs = new STRLSTG[state.getSubnodes().size()];

        if (stgs.length < 1) {
            throw new STRLNotSupportedException("ANDState␣" + state.getID()
                    + "␣without␣a␣region");
        }

        RegionConverter regionConv = new RegionConverter(this);

        for (int i = 0; i < stgs.length; i++) {
            Object nextRegion = regionIterator.next();
            if (nextRegion instanceof Region) {
                stgs[i] = regionConv.convert((Region) nextRegion);
            } else {
                throw new STRLNotSupportedException("cannot␣use␣" + nextRegion
                        + "␣instead␣of␣region␣in␣" + state.getID());
            }
        }

        boolean temp = stgs[0].hasDeepHistory();

        for (int i = 1; i < stgs.length; i++) {
            if (temp != stgs[i].hasDeepHistory()) {
                throw new STRLNotSupportedException("inconsistent␣history␣in␣"
                        + state.getID());
            }
        }

        result.setDeepHistory(temp);

        if (state instanceof FinalANDState) {
            result.setFinal();
        }

        result.setSTGs(stgs);

        return result;
    }
}
```

.6.4 CompConverter.java

```java
/*
 * Created on 25.06.2004
 */
package kiel.simulator.syncchart.converter;

import kiel.dataStructure.boolexp.BooleanComparator;
import kiel.dataStructure.boolexp.Equal;
import kiel.dataStructure.boolexp.GreaterOrEqual;
import kiel.dataStructure.boolexp.GreaterThan;
import kiel.dataStructure.boolexp.LessOrEqual;
import kiel.dataStructure.boolexp.LessThan;
import kiel.dataStructure.boolexp.NotEqual;
//import kiel.dataStructure.boolexp.BooleanCompNE;
import kiel.simulator.syncchart.boolexp.STRLComparator;
import kiel.simulator.syncchart.boolexp.STRLEqual;
import kiel.simulator.syncchart.boolexp.STRLGreaterOrEqual;
import kiel.simulator.syncchart.boolexp.STRLGreaterThan;
import kiel.simulator.syncchart.boolexp.STRLLessOrEqual;
import kiel.simulator.syncchart.boolexp.STRLLessThan;
import kiel.simulator.syncchart.boolexp.STRLNotEqual;
import kiel.simulator.syncchart.exceptions.STRLException;
import kiel.simulator.syncchart.exceptions.STRLNotSupportedException;

/**
 * CompConverter converts kiel BooleanComparator to STRLComparator.
 * @author Andre Ohlhoff
 */
public class CompConverter extends Context {

    /**
     * Constructs a CompConverter.
     * @param context
     *            context for debug messages and signal/variable searching.
     */
    public CompConverter(final Context context) {
        super(context);
    }

    /**
     * Converts a kiel BooleanComparator to a STRLComparator.
     * @param comp
     *            kiel BooleanComparator
     * @return STRLComparator
     * @throws STRLException
     *            The exception is thrown if something is not convertable.
     */
    public final STRLComparator convert(final BooleanComparator comp)
            throws STRLException {
        IntExpConverter ieConv = new IntExpConverter(this);
        if (comp instanceof Equal) {
            STRLEqual result = new STRLEqual(ieConv.convert(comp.getLeftExp()),
                    ieConv.convert(comp.getRightExp()));
            return result;
        } else if (comp instanceof GreaterOrEqual) {
            STRLGreaterOrEqual result = new STRLGreaterOrEqual(ieConv
                    .convert(comp.getLeftExp()), ieConv.convert(comp
                    .getRightExp()));
            return result;
        } else if (comp instanceof GreaterThan) {
            STRLGreaterThan result = new STRLGreaterThan(ieConv.convert(comp
                    .getLeftExp()), ieConv.convert(comp.getRightExp()));
            return result;
        } else if (comp instanceof LessOrEqual) {
            STRLLessOrEqual result = new STRLLessOrEqual(ieConv.convert(comp
                    .getLeftExp()), ieConv.convert(comp.getRightExp()));
            return result;
        } else if (comp instanceof LessThan) {
            STRLLessThan result = new STRLLessThan(ieConv.convert(comp
                    .getLeftExp()), ieConv.convert(comp.getRightExp()));
            return result;
        } else if (comp instanceof NotEqual) {
            STRLNotEqual result = new STRLNotEqual(ieConv.convert(comp
                    .getLeftExp()), ieConv.convert(comp.getRightExp()));
            return result;
        } else {
            throw new STRLNotSupportedException("cannot␣use␣" + comp
                    + "␣as␣comparator");
        }
    }
}
```

.6.5 Context.java

```
/*
 * Created on 18.06.2004
 */
package kiel.simulator.syncchart.converter;

import java.util.Collection;
import java.util.HashSet;
import java.util.Hashtable;

import kiel.dataStructure.Variable;
import kiel.dataStructure.eventexp.CombineWithAdd;
import kiel.dataStructure.eventexp.CombineWithMult;
import kiel.dataStructure.eventexp.Event;
import kiel.dataStructure.eventexp.IntegerSignal;
import kiel.dataStructure.eventexp.Signal;
import kiel.dataStructure.eventexp.TrueSignal;
import kiel.dataStructure.intexp.IntegerExpression;
import kiel.dataStructure.intexp.IntegerVariable;
import kiel.simulator.syncchart.exceptions.STRLException;
import kiel.simulator.syncchart.exceptions.STRLIntegerSignalNotFoundException;
import kiel.simulator.syncchart.exceptions.STRLNotSupportedException;
import kiel.simulator.syncchart.exceptions.STRLSignalNotFoundException;
import kiel.simulator.syncchart.exceptions.STRLVariableNotFoundException;
import kiel.simulator.syncchart.intexp.STRLIntegerExpression;
import kiel.simulator.syncchart.intexp.STRLIntegerVariable;
import kiel.simulator.syncchart.sigexp.STRLIntegerSignal;
import kiel.simulator.syncchart.sigexp.STRLCombineWithAdd;
import kiel.simulator.syncchart.sigexp.STRLCombineWithMult;
import kiel.simulator.syncchart.sigexp.STRLPureSignal;
import kiel.simulator.syncchart.sigexp.STRLSignal;
import kiel.simulator.syncchart.sigexp.STRLSignalTrue;
import kiel.simulator.syncchart.sigexp.STRLSingleSignal;
import kiel.util.LogFile;

/**
 * Context is the basic class for all converter but also for the simulator. A
 * context is used to generate debug messages, but more importent to search
 * signals and variables. When a converter wants to convert some kind of
 * expression, it needs a reference to the corresponding signal or variable
 * which might be defined in a superordinated STG or MacroState.
 * @author Andre Ohlhoff
 */
public abstract class Context {

    /**
     * loglevel for warnings.
     */
    public static final int WARNING = 3;

    /**
     * loglevel for 'normal' simulator output.
     */
    public static final int SIM = 2;

    /**
     * loglevel for debug messages.
     */
    public static final int DEBUG = 1;

    /**
     * maps between kiel events and STRLSignals.
     */
    private Hashtable localSigs = new Hashtable();

    /**
     * maps between kiel variables and STRLIntegerVariables.
     */
    private Hashtable localVars = new Hashtable();

    /**
     * collection of all signals used in the statechart.
     */
    private Collection allSignals = null;

    /**
     * collection of all variables used in the statechart.
     */
    private Collection allVariables = null;

    /**
     * context of this context.
     */
    private Context context = null;

    /**
     * logfile for output.
     */
    private LogFile log = null;

    /**
     * Constructs a context with a superordinated context.
     * @param c another context
     */
    public Context(final Context c) {
        this.context = c;
    }

    /**
     * Constructs the outermost context, which has no context by itself.
     */
    public Context() {
        this.allSignals = new HashSet();
        this.allVariables = new HashSet();
    }

    /**
     * Sets the logfile. Do this only for the outermost context.
     * @param l logfile
     */
    public final void setLogFile(final LogFile l) {
        this.log = l;
    }
```

```
/**
 * Adds the signal to the collection of the outermost context.
 * @param signal new signal
 */
public final void addSignal(final STRLSignal signal) {
    if (this.allSignals == null) {
        this.context.addSignal(signal);
    } else {
        this.allSignals.add(signal);
    }
}

/**
 * Adds the variable to the collection of the outermost context.
 * @param variable new variable
 */
public final void addVariable(final STRLIntegerVariable variable) {
    if (this.allVariables == null) {
        this.context.addVariable(variable);
    } else {
        this.allVariables.add(variable);
    }
}

/**
 * Returns all signals.
 * @return Returns all signals.
 */
public final Collection getSignals() {
    return this.allSignals;
}

/**
 * Returns all variables.
 * @return Returns all variables.
 */
public final Collection getVariables() {
    return this.allVariables;
}

/**
 * Clears the collection of signals and variables.
 */
public final void clear() {
    this.allSignals.clear();
    this.allVariables.clear();
    // this.allSignals = new HashSet();
    // this.allVariables = new HashSet();
}

/**
 * Returns the corresponding STRLSignal for the given kiel signal.
 * @param s kiel signal
 * @return STRLSignal
 * @throws STRLSignalNotFoundException The exception is thrown, if the
 *              signal is not defined.
 */
public final STRLPureSignal toSignal(final Signal s)
        throws STRLSignalNotFoundException {
    STRLPureSignal ps = (STRLPureSignal) this.localSigs.get(s);
    if (ps == null) {
        if (this.context != null) {
            return this.context.toSignal(s);
        } else {
            throw new STRLSignalNotFoundException(s);
        }
    } else {
        return ps;
    }
}

/**
 * Returns the corresponding STRLIntegerSignal for the given kiel.
 * IntegerSignal.
 * @param s kiel integerSignal
 * @return STRLIntegerSignal
 * @throws STRLIntegerSignalNotFoundException The exceptions is thrown, if
 *              the signals is not defined.
 */
public final STRLIntegerSignal toIntegerSignal(final IntegerSignal s)
        throws STRLIntegerSignalNotFoundException {
    STRLIntegerSignal is = (STRLIntegerSignal) this.localSigs.get(s);
    if (is == null) {
        if (this.context != null) {
            return this.context.toIntegerSignal(s);
        } else {
            throw new STRLIntegerSignalNotFoundException(s);
        }
    } else {
        return is;
    }
}

/**
 * Returns the corresponding STRLIntegerSignal as STRLIntegerExpression.
 * @param signal kiel integer signal
 * @return STRLIntegerSignal as STRLIntegerExpression.
 * @throws STRLIntegerSignalNotFoundException The exceptions is thrown, if
 *              the signals is not defined.
 */
public final STRLIntegerExpression toIntExp(final IntegerSignal signal)
        throws STRLIntegerSignalNotFoundException {
    return (STRLIntegerExpression) this.toIntegerSignal(signal);
}

/**
 * Returns the corresponding STRLIntegerVariable as STRLIntegerExpression.
 * @param iv kiel integer variable
 * @return Returns the corresponding STRLIntegerVariable as
 *          STRLIntegerExpression.
 * @throws STRLVariableNotFoundException The exception is thrown if the
 *              variable is not defined.
 */
public final STRLIntegerExpression toIntExp(final IntegerVariable iv)
        throws STRLVariableNotFoundException {
    STRLIntegerExpression result = (STRLIntegerExpression) this.localVars
            .get(iv);
    if (result == null) {
        if (this.context != null) {
                return this.context.toIntExp(iv);
            } else {
                throw new STRLVariableNotFoundException(iv);
            }
        } else {
            return result;
        }
    }

    /**
     * Returns the corresponding STRLIntegerVariable.
     * @param iv kiel integer variable
     * @return Returns the corresponding STRLIntegerVariable.
     * @throws STRLVariableNotFoundException The exception is thrown if the
     *             variable is not defined.
     */
    public final STRLIntegerVariable toIntVariable(final IntegerVariable iv)
            throws STRLVariableNotFoundException {
        return (STRLIntegerVariable) this.toIntExp(iv);
    }

    /**
     * Writes a message into the logfile.
     * @param level loglevel
     * @param message message
     */
    public final void log(final int level, final String message) {
        if (context != null) {
            this.context.log(level, message);
        } else {
            this.log.log(level, message);
        }
    }

    /**
     * Writes a simulator message into the logfile.
     * @param message message
     */
    public final void logSim(final String message) {
        this.log(Context.SIM, message);
    }

    /**
     * Writes a debug message into the logfile.
     * @param message message
     */
    public final void debug(final String message) {
        this.log(Context.DEBUG, message);
    }

    /**
     * Writes a warning into the logfile.
     * @param message warning
     */
    public final void warning(final String message) {
        this.log(Context.WARNING, message);
    }

    /**
     * Converst an array of kiel events into an array of STRLSignals.
     * @param events kiel events
     * @return STRLSignals
     * @throws STRLException The exception is thrown if an unsupported event
     *             type is used.
     */
    public final STRLSignal[] convertSignals(final Event[] events)
            throws STRLException {
        STRLSignal[] result = new STRLSignal[events.length];
        for (int i = 0; i < events.length; i++) {
            result[i] = new STRLSignalTrue(new TrueSignal());
            if (events[i] instanceof CombineWithAdd) {
                CombineWithAdd cwa = (CombineWithAdd) events[i];
                if (cwa.isInitialized()) {
                    IntegerExpression ie = cwa.getInitialValue();
                    IntExpConverter iec = new IntExpConverter(this);
                    STRLIntegerExpression strlie = iec.convert(ie);
                    result[i] = new STRLCombineWithAdd(cwa, strlie);
                    this.debug("converted␣combine␣signal:␣"
                            + events[i].getName());

                } else {
                    this.debug("converted␣uninitialized␣combine␣signal:␣"
                            + events[i].getName());
                    result[i] = new STRLCombineWithAdd(cwa);
                }
            } else if (events[i] instanceof CombineWithMult) {
                CombineWithMult cwm = (CombineWithMult) events[i];
                if (cwm.isInitialized()) {
                    IntegerExpression ie = cwm.getInitialValue();
                    IntExpConverter iec = new IntExpConverter(this);
                    STRLIntegerExpression strlie = iec.convert(ie);
                    result[i] = new STRLCombineWithMult(cwm, strlie);
                    this.debug("converted␣combine␣signal:␣"
                            + events[i].getName());
                } else {
                    this.debug("converted␣uninitialized␣combine␣signal:␣"
                            + events[i].getName());
                    result[i] = new STRLCombineWithMult(cwm);
                }
            } else if (events[i] instanceof IntegerSignal) {
                IntegerSignal is = (IntegerSignal) events[i];
                if (is.isInitialized()) {
                    IntegerExpression ie = is.getInitialValue();
                    IntExpConverter iec = new IntExpConverter(this);
                    STRLIntegerExpression strlie = iec.convert(ie);
                    result[i] = new STRLSingleSignal(is, strlie);
                    this.debug("converted␣single␣signal:␣"
                            + events[i].getName());
                } else {
                    this.debug("converted␣uninitialized␣single␣signal:␣"
                            + events[i].getName());
                    result[i] = new STRLSingleSignal(is);
                }
            } else if (events[i] instanceof Signal) {
                result[i] = new STRLPureSignal((Signal) events[i]);
                this.debug("converted␣pure␣signal:␣" + events[i].getName());
            } else {
                throw new STRLNotSupportedException("cannot␣use␣" + events[i]
                        + "as␣signal");
            }
            this.addSignal(result[i]);
            this.localSigs.put(events[i], result[i]);
        }
        return result;
    }

    /**
     * Converts a collection of kiel events into an array of STRLSignals.
     * @param events collection of kiel events.
     * @return STRLSignals
     * @throws STRLException The exception is thrown if an unsupported event
     *                   type is used.
     */
    public final STRLSignal[] convertSignals(final Collection events)
            throws STRLException {
        Event[] e = (Event[]) events.toArray(new Event[events.size()]);
        return this.convertSignals(e);
    }

    /**
     * Converst an array of kiel variables into an array of
     * STRLIntegerVariables.
     * @param vars kiel variables
     * @return STRLVariables
     * @throws STRLException The excpetion is thrown if an unsupported variable
     *                   type is used.
     */
    public final STRLIntegerVariable[] convertVariables(final Variable[] vars)
            throws STRLException {
        STRLIntegerVariable[] result = new STRLIntegerVariable[vars.length];
        for (int i = 0; i < vars.length; i++) {
            if (vars[i] instanceof IntegerVariable) {
                IntegerVariable ivar = (IntegerVariable) vars[i];
                if (ivar.isInitialized()) {
                    IntegerExpression ie = ivar.getInitialValue();
                    IntExpConverter iec = new IntExpConverter(this);
                    STRLIntegerExpression strlie = iec.convert(ie);
                    result[i] = new STRLIntegerVariable(ivar, strlie);
                } else {
                    result[i] = new STRLIntegerVariable(ivar);
                }
            } else {
                throw new STRLNotSupportedException("cannot␣use␣" + vars[i]
                        + "␣as␣variable");
            }
            this.debug("converted␣variable:␣"
                    + result[i].getKielVar().getName());
            this.addVariable(result[i]);
            this.localVars.put(vars[i], result[i]);
        }
        return result;
    }

    /**
     * Converts a collection of kiel variables into STRLIntegerVariables.
     * @param vars collection of kiel variables
     * @return STRLIntegerVariables
     * @throws STRLException The exception is thrown if an unsupported variable
     *                   type is used
     */
    public final STRLIntegerVariable[] convertVariables(final Collection vars)
            throws STRLException {
        Variable[] v = (Variable[]) vars.toArray(new Variable[vars.size()]);
        return this.convertVariables(v);
    }

}
```

.6.6 DelayExpressionConverter.java

```
/*
 * Created on 07.10.2004
 */
package kiel.simulator.syncchart.converter;

import kiel.dataStructure.CompoundLabel;
import kiel.dataStructure.StringLabel;
import kiel.dataStructure.TransitionLabel;
import kiel.dataStructure.eventexp.DelayExpression;
import kiel.simulator.syncchart.exceptions.STRLException;
import kiel.simulator.syncchart.transition.STRLDelayExpression;

/**
 * DelayExpressionConverter convert kiel delay expressions into
 * STRLDelayExpressions.
 * @author Andre Ohlhoff
 */
public class DelayExpressionConverter extends Context {

    /**
     * Constructs a DelayExpressionConverter.
     * @param c
     *            context for debug messages and signal/variable searching.
     */
    public DelayExpressionConverter(final Context c) {
        super(c);
    }

    /**
     * Generates a STRLDelayExpression out of a kiel transitionlabel.
     * @param label
     *            kiel transitionlabel
     * @return STRLDelayExpression
     * @throws STRLException
     *             The exception is thrown if something is not convertable.
     */
    public final STRLDelayExpression convert(final TransitionLabel label)
            throws STRLException {

        if (label instanceof CompoundLabel) {
            DelayExpression delay = ((CompoundLabel) label).getTrigger();
            STRLDelayExpression result = new STRLDelayExpression();
            if (delay.isImmediate()) {
                result.setImmediateFlag();
            }
            TriggerConverter triggerConv = new TriggerConverter(this);
            result
                    .setSignalExp(triggerConv.convert(delay
                            .getEventExpression()));
            IntExpConverter intExpConv = new IntExpConverter(this);
            result.setCountDelay(intExpConv.convert(delay.getCountDelay()));
            return result;
        } else if (!((StringLabel) label).toString().equals("")) {
            this.warning("StringLabel:␣\"" + label.toString() + "\"␣ignored");
        }
        return new STRLDelayExpression();
    }
}
```

.6.7 DynChoiceToCondCell.java

```
/*
 * Created on 18.06.2004
 */
package kiel.simulator.syncchart.converter;

import kiel.dataStructure.DynamicChoice;
import kiel.simulator.syncchart.STRLConditionalCell;
import kiel.simulator.syncchart.STRLPseudoCell;

/**
 * DynChoiceToCondCell converts a kiel dynamicchoice pseudostate into a
 * STRLPseudoCell.
 * @author Andre Ohlhoff
 */
public class DynChoiceToCondCell extends Context {

    /**
     * Constructs a DynChoiceToCondCell converter.
     * @param c
     *             context for debug messages and signal/variable searching.
     */
    public DynChoiceToCondCell(final Context c) {
        super(c);
    }

    /**
     * Converts a kiel dynamicChoice pseudostate into a STRLPseudoCell.
     * @param state
     *             kiel dynamicchoice
     * @return STRLPseudoCell
     */
    public final STRLPseudoCell convert(final DynamicChoice state) {
        this.debug("DynamicChoice:␣" + state.getID());
        return new STRLConditionalCell(state);
    }

}
```

.6.8 GuardConverter.java

```
/*
 * Created on 25.06.2004
 */
package kiel.simulator.syncchart.converter;

import kiel.dataStructure.CompoundLabel;
import kiel.dataStructure.StringLabel;
import kiel.dataStructure.TransitionLabel;
import kiel.dataStructure.boolexp.BooleanAnd;
import kiel.dataStructure.boolexp.BooleanBrackets;
import kiel.dataStructure.boolexp.BooleanComparator;
import kiel.dataStructure.boolexp.BooleanExpression;
import kiel.dataStructure.boolexp.BooleanFalse;
import kiel.dataStructure.boolexp.BooleanNot;
import kiel.dataStructure.boolexp.BooleanOr;
import kiel.dataStructure.boolexp.BooleanTrue;
import kiel.dataStructure.boolexp.BooleanTrueDummy;
import kiel.simulator.syncchart.boolexp.STRLAnd;
import kiel.simulator.syncchart.boolexp.STRLBrackets;
import kiel.simulator.syncchart.boolexp.STRLComparator;
import kiel.simulator.syncchart.boolexp.STRLBooleanExpression;
import kiel.simulator.syncchart.boolexp.STRLFalse;
import kiel.simulator.syncchart.boolexp.STRLNot;
import kiel.simulator.syncchart.boolexp.STRLOr;
import kiel.simulator.syncchart.boolexp.STRLTrue;
import kiel.simulator.syncchart.exceptions.STRLException;
import kiel.simulator.syncchart.exceptions.STRLNotSupportedException;

/**
 * GuardConverter converts kiel boolean expression used as guards into
 * STRLBooleanExpressions.
 * @author Andre Ohlhoff
 */
public class GuardConverter extends Context {

    /**
     * Constructs a GuardConverter.
     * @param context context for debug messages and signal/variable searching.
     */
    public GuardConverter(final Context context) {
        super(context);
    }

    /**
     * Generates a STRLBooleanExpression used as guard out of a kiel
     * transitionlabel.
     * @param label kiel transitionlabel
     * @return STRLBooleanExpression
     * @throws STRLException The exception is thrown if something is not
     *             convertable.
     */
    public final STRLBooleanExpression convert(final TransitionLabel label)
            throws STRLException {
        if (label instanceof CompoundLabel) {
            return this.convert(((CompoundLabel) label).getCondition());
        } else if (!((StringLabel) label).toString().equals("")) {
            this.warning("WARNING:␣StringLabel:␣\"" + label.toString()
                    + "\"␣ignored");
        }

        return new STRLTrue();

    }

    /**
     * Converts a kiel boolean expression into a STRLBooleanExpression.
     * @param c kiel boolean expression
     * @return STRLBooleanExpression
     * @throws STRLException The exception is thrown if something is not
     *             convertable.
     */
    public final STRLBooleanExpression convert(final BooleanExpression c)
            throws STRLException {
        if (c instanceof BooleanAnd) {
            BooleanAnd and = (BooleanAnd) c;
            STRLAnd result = new STRLAnd(this.convert(and.getLeft()), this
                    .convert(and.getRight()));
            return result;
        } else if (c instanceof BooleanTrueDummy) {
            return new STRLTrue();
        } else if (c instanceof BooleanBrackets) {
            BooleanBrackets bb = (BooleanBrackets) c;
            STRLBrackets result = new STRLBrackets(this.convert(bb.getBody()));
            return result;
        } else if (c instanceof BooleanTrue) {
            return new STRLTrue();
        } else if (c instanceof BooleanFalse) {
            return new STRLFalse();
        } else if (c instanceof BooleanNot) {
            BooleanNot not = (BooleanNot) c;
            STRLNot result = new STRLNot(this.convert(not.getBody()));
            return result;
        } else if (c instanceof BooleanOr) {
            BooleanOr or = (BooleanOr) c;
            STRLOr result = new STRLOr(this.convert(or.getLeft()), this
                    .convert(or.getRight()));
            return result;
        } else if (c instanceof BooleanComparator) {
            CompConverter cConv = new CompConverter(this);
            STRLComparator result = cConv.convert((BooleanComparator) c);
            return result;
        } else {
            throw new STRLNotSupportedException("cannot␣use␣" + c
                    + "␣as␣boolean␣expression");
        }
    }
}
```

.6.9 ImmediateTransConv.java

```
/*
 * Created on 20.06.2004
 */
package kiel.simulator.syncchart.converter;

import java.util.Hashtable;

import kiel.dataStructure.Transition;
import kiel.simulator.syncchart.exceptions.STRLException;
import kiel.simulator.syncchart.transition.STRLImmediateTrans;

/**
 * ImmediateTransConv converts kiel transitions into STRLImmediateTransitions.
 * @author Andre Ohlhoff
 */
public class ImmediateTransConv extends TransConverter {

    /**
     * Constructs a ImmediateTransConv.
     * @param context
     *            context for debug messages and signal/variable searching.
     */
    public ImmediateTransConv(final Context context) {
        super(context);
    }

    /**
     * Converts a kiel transition into a STRLImmediateTransition.
     * @param t
     *            kiel transition
     * @param celltable
     *            mapping between kiel nodes und syncchart cells.
     * @return STRLImmediateTransition
     * @throws STRLException
     *             The exception is thrown if something is not convertable.
     */
    public final STRLImmediateTrans convert(final Transition t,
            final Hashtable celltable) throws STRLException {
        STRLImmediateTrans it = new STRLImmediateTrans(t);
        return (STRLImmediateTrans) this.convert(it, celltable);
    }
}
```

.6.10 InitStateToInitCell.java

```java
/*
 * Created on 18.06.2004
 */
package kiel.simulator.syncchart.converter;

import kiel.dataStructure.InitialState;
import kiel.simulator.syncchart.STRLInitialCell;

/**
 * InitStateToInitCell converts kiel InitialStates into STRLInitialCells.
 * @author Andre Ohlhoff
 */
public class InitStateToInitCell extends Context {

    /**
     * Constructs a InitStateToInitCell converter.
     * @param c context for debug messages and signal/variable searching.
     */
    public InitStateToInitCell(final Context c) {
        super(c);
    }

    /**
     * Converts a kiel InitialState into a STRLInitialCell.
     * @param state kiel state
     * @return STRLInitialCell
     */
    public final STRLInitialCell convert(final InitialState state) {
        this.debug("InitialState:␣" + state.getID());
        return new STRLInitialCell(state);
    }

}
```

.6.11 IntExpConverter.java

```
/*
 * Created on 25.06.2004
 */
package kiel.simulator.syncchart.converter;

import kiel.dataStructure.boolexp.BooleanExpression;
import kiel.dataStructure.boolexp.Pre;
import kiel.dataStructure.eventexp.IntegerSignal;
import kiel.dataStructure.intexp.IntegerAdd;
import kiel.dataStructure.intexp.IntegerBrackets;
import kiel.dataStructure.intexp.IntegerConstant;
import kiel.dataStructure.intexp.IntegerDiv;
import kiel.dataStructure.intexp.IntegerExpression;
import kiel.dataStructure.intexp.IntegerMod;
import kiel.dataStructure.intexp.IntegerMult;
import kiel.dataStructure.intexp.IntegerMinus;
import kiel.dataStructure.intexp.IntegerSub;
import kiel.dataStructure.intexp.IntegerVariable;
import kiel.simulator.syncchart.exceptions.STRLException;
import kiel.simulator.syncchart.exceptions.STRLNotSupportedException;
import kiel.simulator.syncchart.intexp.STRLIntegerAdd;
import kiel.simulator.syncchart.intexp.STRLIntegerBrackets;
import kiel.simulator.syncchart.intexp.STRLIntegerConstant;
import kiel.simulator.syncchart.intexp.STRLIntegerDiv;
import kiel.simulator.syncchart.intexp.STRLIntegerExpression;
import kiel.simulator.syncchart.intexp.STRLIntegerMod;
import kiel.simulator.syncchart.intexp.STRLIntegerMult;
import kiel.simulator.syncchart.intexp.STRLIntegerMinus;
import kiel.simulator.syncchart.intexp.STRLIntegerSub;
import kiel.simulator.syncchart.sigexp.STRLPre;

/**
 * IntExpConverter converts kiel IntegerExpression into STRLIntegerExpressions.
 * @author Andre Ohlhoff
 */
public class IntExpConverter extends Context {

    /**
     * Constructs a IntExpConverter.
     * @param context
     *            context for debug messages and signal/variable searching.
     */
    public IntExpConverter(final Context context) {
        super(context);
    }

    /**
     * Converts a kiel IntegerExpression into a STRLIntegerExpression.
     * @param exp
     *            kiel IntegerExpression
     * @return STRLIntegerExpression
     * @throws STRLException
     *             The exception is thrown if something is not convertable.
     */
    public final STRLIntegerExpression convert(final IntegerExpression exp)
            throws STRLException {
        if (exp instanceof IntegerConstant) {
            return new STRLIntegerConstant(((IntegerConstant) exp).getValue());
        } else if (exp instanceof IntegerBrackets) {
            IntegerBrackets be = (IntegerBrackets) exp;
            return new STRLIntegerBrackets(this.convert(be.getBody()));
        } else if (exp instanceof IntegerMinus) {
            IntegerMinus intNeg = (IntegerMinus) exp;
            return new STRLIntegerMinus(this.convert(intNeg.getBody()));
        } else if (exp instanceof IntegerAdd) {
            IntegerAdd intAdd = (IntegerAdd) exp;
            return new STRLIntegerAdd(this.convert(intAdd.getLeftExp()), this
                    .convert(intAdd.getRightExp()));
        } else if (exp instanceof IntegerDiv) {
            IntegerDiv intDiv = (IntegerDiv) exp;
            return new STRLIntegerDiv(this.convert(intDiv.getLeftExp()), this
                    .convert(intDiv.getRightExp()));
        } else if (exp instanceof IntegerMod) {
            IntegerMod intMod = (IntegerMod) exp;
            return new STRLIntegerMod(this.convert(intMod.getLeftExp()), this
                    .convert(intMod.getRightExp()));
        } else if (exp instanceof IntegerMult) {
            IntegerMult intMult = (IntegerMult) exp;
            return new STRLIntegerMult(this.convert(intMult.getLeftExp()), this
                    .convert(intMult.getRightExp()));
        } else if (exp instanceof IntegerSub) {
            IntegerSub intSub = (IntegerSub) exp;
            return new STRLIntegerSub(this.convert(intSub.getLeftExp()), this
                    .convert(intSub.getRightExp()));
        } else if (exp instanceof IntegerSignal) {
            return this.toIntExp((IntegerSignal) exp);
        } else if (exp instanceof IntegerVariable) {
            return this.toIntExp((IntegerVariable) exp);
        } else if (exp instanceof Pre) {
            STRLPre p = new STRLPre();
            BooleanExpression be = ((Pre) exp).getExpression();
            if (be instanceof IntegerSignal) {
                p.setIntSig(this.toIntegerSignal((IntegerSignal) be));
                return p;
            } else {
                throw new STRLNotSupportedException("cannot␣use␣" + be
                        + "␣in␣pre␣in␣an␣integer␣context");
            }
        } else {
            throw new STRLNotSupportedException("cannot␣use␣" + exp
                    + "␣as␣integer␣expression");
        }
    }
}
```

.6.12 KielToSyncChart.java

```
/*
 * Created on 18.06.2004
 */
package kiel.simulator.syncchart.converter;

import kiel.dataStructure.ANDState;
import kiel.dataStructure.Node;
import kiel.dataStructure.ORState;
import kiel.dataStructure.State;
import kiel.dataStructure.StateChart;
import kiel.simulator.syncchart.STRLMacroState;
import kiel.simulator.syncchart.SyncChart;
import kiel.simulator.syncchart.exceptions.STRLException;
import kiel.simulator.syncchart.exceptions.STRLNotSupportedException;

/**
 * KielToSyncChart converts a kiel statechart into a syncchart with same
 * topology.
 * @author Andre Ohlhoff
 */
public class KielToSyncChart extends Context {

    /**
     * Constructs a KielToSyncChart converter.
     * @param context
     *            context for debug messages and signal/variable searching.
     */
    public KielToSyncChart(final Context context) {
        super(context);
    }

    /**
     * Converts the given kiel StateChart into a SyncChart.
     * @param statechart
     *            kiel StateChart
     * @return SyncChart
     * @throws STRLException
     *             The exception is thrown if something is not convertable.
     */
    public final SyncChart convert(final StateChart statechart)
            throws STRLException {
        SyncChart syncchart = new SyncChart();

        syncchart.setInput(this.convertSignals(statechart.getInputEvents()));
        syncchart.setOutput(this.convertSignals(statechart.getOutputEvents()));

        STRLMacroState top = null;
        Node root = statechart.getRootNode();

        if (root instanceof ANDState) {
            ANDToMacro converter = new ANDToMacro(this);
            top = converter.convert((ANDState) root);

        } else if (root instanceof ORState) {
            ORToMacro converter = new ORToMacro(this);
            top = converter.convert((ORState) root);
        } else {
            throw new STRLNotSupportedException("cannot␣use␣" + root
                    + "␣as␣root");
        }

        ActionsConverter aconv = new ActionsConverter(this);
        top.setOnInside(aconv.convert(((State) root).getDoActivity()));

        syncchart.setTop(top);

        return syncchart;
    }
}
```

.6.13 NormTermConv.java

```
/*
 * Created on 20.06.2004
 */
package kiel.simulator.syncchart.converter;

import java.util.Hashtable;

import kiel.dataStructure.Transition;
import kiel.simulator.syncchart.exceptions.STRLException;
import kiel.simulator.syncchart.transition.STRLNormalTermination;

/**
 * NormTermConv converts kiel transitions into STRLNormalTerminations.
 * @author Andre Ohlhoff
 */
public class NormTermConv extends TransConverter {

    /**
     * Constructs a NormTermConv.
     * @param context
     *            context for debug messages and signal/variable searching.
     */
    public NormTermConv(final Context context) {
        super(context);
    }

    /**
     * Converts a kiel transition into a STRLNormalTermination.
     * @param t
     *            kiel transition
     * @param celltable
     *            mapping between kiel nodes und syncchart cells.
     * @return STRLImmediateTransition
     * @throws STRLException
     *             The exception is thrown if something is not convertable.
     */
    public final STRLNormalTermination convert(final Transition t,
            final Hashtable celltable) throws STRLException {
        STRLNormalTermination nt = new STRLNormalTermination(t);
        return (STRLNormalTermination) this.convert(nt, celltable);
    }
}
```

.6.14 ORToMacro.java

```
/*
 * Created on 18.06.2004
 */
package kiel.simulator.syncchart.converter;

import kiel.dataStructure.FinalORState;
import kiel.dataStructure.ORState;
import kiel.dataStructure.Region;
import kiel.simulator.syncchart.STRLMacroState;
import kiel.simulator.syncchart.STRLSTG;
import kiel.simulator.syncchart.exceptions.STRLException;

/**
 * ORToMacro converts kiel ORStates into STRLMacroStates.
 * @author Andre Ohlhoff
 */
public class ORToMacro extends Context {

    /**
     * Constructs a ORToMacro converter.
     * @param context
     *            context for debug messages and signal/variable searching.
     */
    public ORToMacro(final Context context) {
        super(context);
    }

    /**
     * Converts a kiel ORState into a STRLMacroState.
     * @param state
     *            kiel ORState
     * @return STRLMacrostate
     * @throws STRLException
     *             The exception is thrown if something is not convertable.
     */
    public final STRLMacroState convert(final ORState state)
            throws STRLException {
        this.debug("ORState:␣" + state.getID());

        STRLMacroState result = new STRLMacroState(state);
        result.setLocalSignals(this.convertSignals(state.getLocalEvents()));

        RegionConverter regionConv = new RegionConverter(this);

        Region temp = new Region();
        temp.addSubnodes(state.getSubnodes());
        temp.addVariables(state.getVariables());

        STRLSTG stg = regionConv.convert(temp);

        result.setDeepHistory(stg.hasDeepHistory());

        result.setSTGs(new STRLSTG[] {stg });

        if (state instanceof FinalORState) {
            result.setFinal();
        }

        return result;
    }
}
```

.6.15 RegionConverter.java

```
/*
 * Created on 06.10.2004
 */
package kiel.simulator.syncchart.converter;

import java.util.ArrayList;
import java.util.Collection;
import java.util.Comparator;
import java.util.Hashtable;
import java.util.Iterator;
import java.util.TreeSet;

import kiel.dataStructure.DeepHistory;
import kiel.dataStructure.DynamicChoice;
import kiel.dataStructure.InitialState;
import kiel.dataStructure.Node;
import kiel.dataStructure.NormalTermination;
import kiel.dataStructure.Region;
import kiel.dataStructure.State;
import kiel.dataStructure.StrongAbortion;
import kiel.dataStructure.Suspend;
import kiel.dataStructure.Transition;
import kiel.dataStructure.WeakAbortion;
import kiel.simulator.syncchart.STRLInitialCell;
import kiel.simulator.syncchart.STRLPseudoCell;
import kiel.simulator.syncchart.STRLReactiveCell;
import kiel.simulator.syncchart.STRLSTG;
import kiel.simulator.syncchart.STRLStateCell;
import kiel.simulator.syncchart.STRLSuspend;
import kiel.simulator.syncchart.exceptions.STRLException;
import kiel.simulator.syncchart.exceptions.STRLNotSupportedException;
import kiel.simulator.syncchart.transition.STRLDelayExpression;
import kiel.simulator.syncchart.transition.STRLImmediateTrans;
import kiel.simulator.syncchart.transition.STRLNormalTermination;
import kiel.simulator.syncchart.transition.STRLStrongAbortion;
import kiel.simulator.syncchart.transition.STRLWeakAbortion;

/**
 * RegionConverter convert kiel Regions into STRLSTGs.
 * @author Andre Ohlhoff
 */
public class RegionConverter extends Context {

    /**
     * Constructs a RegionConverter.
     * @param c
     *            context for debug messages and signal/variable searching.
     */
    public RegionConverter(final Context c) {
        super(c);
    }

    /**
     * Converts a kiel Region into a STRLSTG.
     * @param region
     *            kiel Region
     * @return STRLSTG
     * @throws STRLException
     *             The exception is thrown if something is not convertable.
     */
    public final STRLSTG convert(final Region region) throws STRLException {
        STRLSTG result = new STRLSTG();
        result.setVariables(this.convertVariables(region.getVariables()));

        Collection cells = new ArrayList();

        Iterator nodeIterator = region.getSubnodes().iterator();

        boolean hasInit = false;

        // alle Nodes einzeln transformieren
        while (nodeIterator.hasNext()) {
            Node nextNode = (Node) nodeIterator.next();
            if (nextNode instanceof DynamicChoice) {
                DynChoiceToCondCell converter = new DynChoiceToCondCell(this);
                cells.add(converter.convert((DynamicChoice) nextNode));
            } else if (nextNode instanceof InitialState) {
                hasInit = true;
                InitStateToInitCell converter = new InitStateToInitCell(this);
                cells.add(converter.convert((InitialState) nextNode));
            } else if (nextNode instanceof DeepHistory) {
                result.setDeepHistory();
            } else if (nextNode instanceof State) {
                StateToStateCell converter = new StateToStateCell(this);
                cells.add(converter.convert((State) nextNode));
            } else if (nextNode instanceof Suspend) {
                this.debug("found␣suspension");
            } else {
                throw new STRLNotSupportedException("cannot␣use␣" + nextNode
                        + "␣in␣region");
            }
        }

        if (!hasInit) {
            throw new STRLException("region␣without␣initial␣state");
        }

        STRLReactiveCell[] recell = (STRLReactiveCell[]) cells
                .toArray(new STRLReactiveCell[cells.size()]);
        result.setCells(recell);

        this.computeTransitions(recell);
        return result;
    }

    /**
     * Generates STRLTransitions between the given cells.
     * @param cells
     *            STRLReactiveCells without transitions.
     * @throws STRLException
     *             The exception is thrown if something is not convertable.
     */
```

```
private void computeTransitions(final STRLReactiveCell[] cells)
        throws STRLException {

    // Tabelle fÃ¼r den Zugriff: Kiel Node -> ReactiveCell
    Hashtable celltable = new Hashtable();
    for (int i = 0; i < cells.length; i++) {
        celltable.put(cells[i].getKielNode(), cells[i]);
    }

    for (int i = 0; i < cells.length; i++) {
        TreeSet ts = new TreeSet(new Comparator() {
            public int compare(final Object o1, final Object o2) {
                Transition t1 = (Transition) o1;
                Transition t2 = (Transition) o2;
                return t1.getPriority().getValue()
                        - t2.getPriority().getValue();
            }
        });
        ts.addAll(cells[i].getKielNode().getOutgoingTransitions());

        if (cells[i] instanceof STRLStateCell) {
            STRLStateCell c = (STRLStateCell) cells[i];

            // eingehende Transition auf Suspend untersuchen
            Collection incoming = c.getKielNode().getIncomingTransitions();
            Iterator inIterator = incoming.iterator();
            while (inIterator.hasNext()) {
                Transition t = (Transition) inIterator.next();
                if (t.getSource() instanceof Suspend) {
                    DelayExpressionConverter dEConv
                    = new DelayExpressionConverter(this);
                    STRLDelayExpression de = dEConv.convert(t.getLabel());
                    STRLSuspend s = new STRLSuspend();
                    if (de.isImmediate()) {
                        s.setImmediate();
                    }
                    s.setSigExp(de.getExp());
                    c.setSuspension(s);
                    this.debug("added␣suspension:␣" + t);
                }
            }

            // ausgehende Transitionen aufteilen
            ArrayList strong = new ArrayList();
            ArrayList weak = new ArrayList();
            ArrayList normterm = new ArrayList();

            Iterator transIterator = ts.iterator();

            while (transIterator.hasNext()) {
                Transition t = (Transition) transIterator.next();
                if (t instanceof WeakAbortion) {
                    weak.add(t);
                } else if (t instanceof StrongAbortion) {
                    strong.add(t);
                } else if (t instanceof NormalTermination) {
                    normterm.add(t);
                }
            }

            STRLStrongAbortion[] sa = new STRLStrongAbortion[strong.size()];
            for (int j = 0; j < sa.length; j++) {
                Transition t = (Transition) strong.get(j);
                this.debug("StrongAbortion:␣" + t);
                StrongAbortConv converter = new StrongAbortConv(this);
                sa[j] = converter.convert(t, celltable);
            }

            STRLWeakAbortion[] wa = new STRLWeakAbortion[weak.size()];
            for (int j = 0; j < wa.length; j++) {
                Transition t = (Transition) weak.get(j);
                this.debug("WeakAbortion:␣" + t);
                WeakAbortConv converter = new WeakAbortConv(this);
                wa[j] = converter.convert(t, celltable);
            }

            STRLNormalTermination[] nt = new STRLNormalTermination[normterm
                    .size()];
            for (int j = 0; j < nt.length; j++) {
                Transition t = (Transition) normterm.get(j);
                this.debug("NormTerm:␣" + t);
                NormTermConv converter = new NormTermConv(this);
                nt[j] = converter.convert(t, celltable);
            }

            int temp = sa.length + wa.length + nt.length;

            if ((c instanceof STRLStateCell)
                    && ((STRLStateCell) c).getBody().isFinal()
                    && (temp > 0)) {
                throw new STRLNotSupportedException(
                        "final␣state␣with␣outgoing␣transitions");
            }

            c.setStrongAborts(sa);
            c.setWeakAborts(wa);
            c.setNormTerms(nt);

        } else if (cells[i] instanceof STRLInitialCell) {
            STRLInitialCell c = (STRLInitialCell) cells[i];
            STRLImmediateTrans[] itrans = new STRLImmediateTrans[ts.size()];
            Iterator transIterator = ts.iterator();
            int j = 0;
            while (transIterator.hasNext()) {
                Transition t = (Transition) transIterator.next();
                this.debug("InitialTrans:␣" + t);
                ImmediateTransConv converter = new ImmediateTransConv(this);
                itrans[j] = converter.convert(t, celltable);
                j++;
            }
            if (itrans.length < 1) {
                throw new STRLNotSupportedException(
                        "intial␣state␣without␣transitions");
            }
            c.setTrans(itrans);

        } else if (cells[i] instanceof STRLPseudoCell) {
            STRLPseudoCell c = (STRLPseudoCell) cells[i];
            STRLImmediateTrans[] ctrans = new STRLImmediateTrans[ts.size()];
            Iterator transIterator = ts.iterator();
```

```
                int j = 0;
                while (transIterator.hasNext()) {
                    Transition t = (Transition) transIterator.next();
                    this.debug("ConditionalTrans:␣" + t);
                    ImmediateTransConv converter = new ImmediateTransConv(this);
                    ctrans[j] = converter.convert(t, celltable);
                    j++;
                }
                if (ctrans.length < 1) {
                    throw new STRLNotSupportedException(
                            "conditinal␣state␣without␣transitions");
                }
                c.setTrans(ctrans);
            }
        }
    }
}
```

.6.16 SimpleToSimple.java

```
/*
 * Created on 18.06.2004
 *
 */
package kiel.simulator.syncchart.converter;

import kiel.dataStructure.FinalSimpleState;
import kiel.dataStructure.SimpleState;
import kiel.simulator.syncchart.STRLSimpleState;

/**
 * SimpleToSimple converts kiel SimpleStates into STRLSimpleStates.
 * @author Andre Ohlhoff
 */
public class SimpleToSimple extends Context {

    /**
     * Constructs a SimpleToSimple converter.
     * @param c
     *            context for debug messages and signal/variable searching.
     */
    public SimpleToSimple(final Context c) {
        super(c);
    }

    /**
     * Converts a kiel SimpleState into a STRLSimpleState.
     * @param state
     *            kiel SimpleState
     * @return STRLSimpleState
     */
    public final STRLSimpleState convert(final SimpleState state) {
        this.debug("SimpleState:␣" + state.getName());
        STRLSimpleState result = new STRLSimpleState(state);
        if (state instanceof FinalSimpleState) {
            result.setFinal();
        }
        return result;
    }

}
```

.6.17 StateToStateCell.java

```java
/*
 * Created on 18.06.2004
 */
package kiel.simulator.syncchart.converter;

import kiel.dataStructure.ANDState;
import kiel.dataStructure.ORState;
import kiel.dataStructure.SimpleState;
import kiel.dataStructure.State;
import kiel.simulator.syncchart.STRLState;
import kiel.simulator.syncchart.STRLStateCell;
import kiel.simulator.syncchart.exceptions.STRLException;
import kiel.simulator.syncchart.exceptions.STRLNotSupportedException;

/**
 * StateToStateCell converts kiel States into STRLStateCells.
 * @author Andre Ohlhoff
 */
public class StateToStateCell extends Context {

    /**
     * Constructs a StateToStateCell converter.
     * @param context
     *            context for debug messages and signal/variable searching.
     */
    public StateToStateCell(final Context context) {
        super(context);
    }

    /**
     * Converts a kiel State into a STRLStateCell.
     * @param state
     *            kiel State
     * @return STRLStateCell.
     * @throws STRLException
     *             The exception is thrown if something is not convertable.
     */
    public final STRLStateCell convert(final State state) throws STRLException {
        STRLStateCell result = new STRLStateCell(state);
        STRLState body = null;

        if (state instanceof ORState) {
            ORToMacro converter = new ORToMacro(this);
            body = converter.convert((ORState) state);
        } else if (state instanceof ANDState) {
            ANDToMacro converter = new ANDToMacro(this);
            body = converter.convert((ANDState) state);
        } else if (state instanceof SimpleState) {
            SimpleToSimple converter = new SimpleToSimple(this);
            body = converter.convert((SimpleState) state);
        } else {
            throw new STRLNotSupportedException("cannot␣use␣" + state
                    + "␣as␣state");
        }

        result.setBody(body);

        ActionsConverter aConv = new ActionsConverter(this);

        result.setOnEntry(aConv.convert(state.getEntry()));
        body.setOnInside(aConv.convert(state.getDoActivity()));
        result.setOnExit(aConv.convert(state.getExit()));

        return result;
    }

}
```

.6.18 StrongAbortConv.java

```java
/*
 * Created on 20.06.2004
 */
package kiel.simulator.syncchart.converter;

import java.util.Hashtable;

import kiel.dataStructure.Transition;
import kiel.simulator.syncchart.exceptions.STRLException;
import kiel.simulator.syncchart.transition.STRLStrongAbortion;

/**
 * StrongAbortionConv converts kiel transitions into STRLStrongAbortions.
 * @author Andre Ohlhoff
 */
public class StrongAbortConv extends TransConverter {

    /**
     * Constructs a StrongAbortionConv.
     * @param context
     *            context for debug messages and signal/variable searching.
     */
    public StrongAbortConv(final Context context) {
        super(context);
    }

    /**
     * Converts a kiel transition into a STRLStrongAbortion.
     * @param t
     *            kiel transition
     * @param celltable
     *            mapping between kiel nodes und syncchart cells.
     * @return STRLStrongAbortion
     * @throws STRLException
     *             The exception is thrown if something is not convertable.
     */
    public final STRLStrongAbortion convert(final Transition t,
            final Hashtable celltable) throws STRLException {
        STRLStrongAbortion sa = new STRLStrongAbortion(t);
        return (STRLStrongAbortion) this.convert(sa, celltable);
    }
}
```

.6.19 TransConverter.java

```
/*
 * Created on 20.06.2004
 */
package kiel.simulator.syncchart.converter;

import java.util.Hashtable;

import kiel.simulator.syncchart.STRLReactiveCell;
import kiel.simulator.syncchart.exceptions.STRLException;
import kiel.simulator.syncchart.transition.STRLTransition;

/**
 * Basic class for all transition converter.
 * @author Andre Ohlhoff
 */
public abstract class TransConverter extends Context {

    /**
     * Constructs a TransConverter.
     * @param context
     *            context for debug messages and signal/variable searching.
     */
    public TransConverter(final Context context) {
        super(context);
    }

    /**
     * Generates missing fields for the given STRLTransition.
     * @param strlt
     *            STRLTransition with empty fields
     * @param celltable
     *            mapping between kiel nodes and STRLReactiveCells.
     * @return Returns a full STRLTransition.
     * @throws STRLException
     *             The exception is thrown if something is not convertable.
     */
    public final STRLTransition convert(final STRLTransition strlt,
            final Hashtable celltable) throws STRLException {
        STRLReactiveCell target = (STRLReactiveCell) celltable.get(strlt
                .getKielTransition().getTarget());
        strlt.setTarget(target);
        strlt.setEntersHistory(target.hasDeepHistory());
        DelayExpressionConverter dEConv = new DelayExpressionConverter(this);
        strlt.setDelayExp(dEConv.convert(strlt.getKielTransition().getLabel()));
        GuardConverter gConv = new GuardConverter(this);
        strlt.setGuard(gConv.convert(strlt.getKielTransition().getLabel()));
        ActionsConverter aConv = new ActionsConverter(this);
        strlt.setActions(aConv.convert(strlt.getKielTransition().getLabel()));
        return strlt;
    }
}
```

.6.20 TriggerConverter.java

```
/*
 * Created on 18.06.2004
 */
package kiel.simulator.syncchart.converter;

import kiel.dataStructure.boolexp.BooleanAnd;
import kiel.dataStructure.boolexp.BooleanBrackets;
import kiel.dataStructure.boolexp.BooleanExpression;
import kiel.dataStructure.boolexp.BooleanNot;
import kiel.dataStructure.boolexp.BooleanOr;
import kiel.dataStructure.boolexp.Pre;
import kiel.dataStructure.eventexp.IntegerSignal;
import kiel.dataStructure.eventexp.Signal;
import kiel.dataStructure.eventexp.Tick;
import kiel.dataStructure.eventexp.TrueSignal;
import kiel.simulator.syncchart.boolexp.STRLAnd;
import kiel.simulator.syncchart.boolexp.STRLBooleanExpression;
import kiel.simulator.syncchart.boolexp.STRLBrackets;
import kiel.simulator.syncchart.boolexp.STRLNot;
import kiel.simulator.syncchart.boolexp.STRLOr;
import kiel.simulator.syncchart.exceptions.STRLException;
import kiel.simulator.syncchart.exceptions.STRLNotSupportedException;
import kiel.simulator.syncchart.sigexp.STRLPre;
import kiel.simulator.syncchart.sigexp.STRLSignalTick;
import kiel.simulator.syncchart.sigexp.STRLSignalTrue;

/**
 * TriggerConverter converts kiel boolean expression used as triggers into
 * STRLBooleanExpressions.
 * @author Andre Ohlhoff
 */
public class TriggerConverter extends Context {

    /**
     * Constructs a TriggerConverter.
     * @param context
     *            context for debug messages and signal/variable searching.
     */
    public TriggerConverter(final Context context) {
        super(context);
    }

    /**
     * Converts a kiel BooleanExpression into a STRLBooleanExpression.
     * @param t
     *            kiel BooleanExpression
     * @return STRLBooleanExpression
     * @throws STRLException
     *             The exception is thrown if something is not convertable.
     */
    public final STRLBooleanExpression convert(final BooleanExpression t)
            throws STRLException {
        this.debug("trigger:␣" + t.toString());

        STRLBooleanExpression result;

        if (t instanceof TrueSignal) {
            result = new STRLSignalTrue((TrueSignal) t);
        } else if (t instanceof Tick) {
            result = new STRLSignalTick((Tick) t);
        } else if (t instanceof IntegerSignal) {
            result = this.toIntegerSignal((IntegerSignal) t);
        } else if (t instanceof Signal) {
            result = this.toSignal((Signal) t);
        } else if (t instanceof BooleanAnd) {
            BooleanAnd ea = (BooleanAnd) t;
            STRLBooleanExpression part1 = this.convert(ea.getLeft());
            STRLBooleanExpression part2 = this.convert(ea.getRight());
            result = new STRLAnd(part1, part2);
        } else if (t instanceof BooleanOr) {
            BooleanOr eo = (BooleanOr) t;
            STRLBooleanExpression part1 = this.convert(eo.getLeft());
            STRLBooleanExpression part2 = this.convert(eo.getRight());
            result = new STRLOr(part1, part2);
        } else if (t instanceof BooleanBrackets) {
            BooleanBrackets eb = (BooleanBrackets) t;
            STRLBooleanExpression body = this.convert(eb.getBody());
            result = new STRLBrackets(body);
        } else if (t instanceof BooleanNot) {
            BooleanNot en = (BooleanNot) t;
            STRLBooleanExpression body = this.convert(en.getBody());
            result = new STRLNot(body);
        } else if (t instanceof Pre) {
            Pre p = (Pre) t;
            STRLBooleanExpression body = this.convert(p.getExpression());
            STRLPre r = new STRLPre();
            r.setExpression(body);
            result = r;
        } else {
            throw new STRLNotSupportedException("cannot␣use␣" + t
                    + "␣as␣trigger");
        }

        return result;
    }
}
```

.6.21 WeakAbortConv.java

```
/*
 * Created on 20.06.2004
 */
package kiel.simulator.syncchart.converter;

import java.util.Hashtable;

import kiel.dataStructure.Transition;
import kiel.simulator.syncchart.exceptions.STRLException;
import kiel.simulator.syncchart.transition.STRLWeakAbortion;

/**
 * WeakAbortConv converts kiel transitions into STRLWeakAbortions.
 * @author Andre Ohlhoff
 */
public class WeakAbortConv extends TransConverter {

    /**
     * Constructs a WeakAbortConv.
     * @param context
     *            context for debug messages and signal/variable searching.
     */
    public WeakAbortConv(final Context context) {
        super(context);
    }

    /**
     * Converts a kiel transition into a STRLWeakAbortion.
     * @param t
     *            kiel transition
     * @param celltable
     *            mapping between kiel nodes und syncchart cells.
     * @return STRLWeakAbortion
     * @throws STRLException
     *             The exception is thrown if something is not convertable.
     */
    public final STRLWeakAbortion convert(final Transition t,
            final Hashtable celltable) throws STRLException {
        STRLWeakAbortion wa = new STRLWeakAbortion(t);
        return (STRLWeakAbortion) this.convert(wa, celltable);
    }
}
```

.7 kiel.simulator.syncchart.exceptions

.7.1 STRLException.java

```
/*
 * Created on 20.10.2004
 */
package kiel.simulator.syncchart.exceptions;

import kiel.simulator.SimulatorException;

/**
 * This is the basic class for all exceptions which occur in the syncchart
 * package.
 * @author Andre Ohlhoff
 */
public class STRLException extends SimulatorException {

    /**
     * UID.
     */
    private static final long serialVersionUID = 1L;

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

}
```

.7.2 STRLIntegerSignalNotFoundException.java

```
/*
 * Created on 20.10.2004
 */
package kiel.simulator.syncchart.exceptions;

import kiel.dataStructure.eventexp.IntegerSignal;

/**
 * An IntegerSignalNotFoundException is thrown, if an integer signal is not
 * found. This may happen during statechart conversion if the signal is not
 * defined correct.
 * @author Andre Ohlhoff
 */
public class STRLIntegerSignalNotFoundException extends STRLException {

    /**
     * UID.
     */
    private static final long serialVersionUID = 1L;

    /**
     * Constructs a exception with the given signals.
     * @param s signal that is not found
     */
    public STRLIntegerSignalNotFoundException(final IntegerSignal s) {
        super("integersignal␣" + s.getName() + "␣not␣found");
    }
}
```

.7.3 STRLNotSupportedException.java

```
/*
 * Created on 20.10.2004
 */
package kiel.simulator.syncchart.exceptions;

/**
 * STRLNotSupportedException is thrown if the statechart contains an component
 * which is not supported of the syncchart simulator.
 * @author Andre Ohlhoff
 */
public class STRLNotSupportedException extends STRLException {

    /**
     * UID.
     */
    private static final long serialVersionUID = 1L;

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

}
```

.7.4 STRLSignalNotFoundException.java

```
/*
 * Created on 20.10.2004
 */
package kiel.simulator.syncchart.exceptions;

import kiel.dataStructure.eventexp.Signal;

/**
 * SignalNotFoundException is thrown, if an signal is not found. This may happen
 * during statechart conversion if the signal is not defined correct.
 * @author Andre Ohlhoff
 */
public class STRLSignalNotFoundException extends STRLException {

    /**
     * UID.
     */
    private static final long serialVersionUID = 1L;

    /**
     * Constructs a exception with the given signals.
     * @param s signal that is not found
     */
    public STRLSignalNotFoundException(final Signal s) {
        super("signal␣" + s.getName() + "␣not␣found");
    }

}
```

.7.5 STRLUninitializedReadException.java

```
/*
 * Created on 20.10.2004
 */
package kiel.simulator.syncchart.exceptions;

import kiel.dataStructure.eventexp.IntegerSignal;
import kiel.dataStructure.intexp.IntegerVariable;

/**
 * An UninitializedReadException is thrown if the value of a signal or a
 * variable is used but not initialized before.
 * @author Andre Ohlhoff
 */
public class STRLUninitializedReadException extends STRLException {

    /**
     * UID.
     */
    private static final long serialVersionUID = 1L;

    /**
     * Constructs an UninitializedReadException with the given signal.
     * @param is signal which is read but not initialized.
     */
    public STRLUninitializedReadException(final IntegerSignal is) {
        super("uninitialized␣read␣of␣signal␣" + is.getName());
    }

    /**
     * Constructs an UninitializedReadException with the given variable.
     * @param iv variable which is read but not initialized.
     */
    public STRLUninitializedReadException(final IntegerVariable iv) {
        super("uninitialized␣read␣of␣variable␣" + iv.getName());
    }

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

}
```

.7.6 STRLVariableNotFoundException.java

```
/*
 * Created on 20.10.2004
 */
package kiel.simulator.syncchart.exceptions;

import kiel.dataStructure.intexp.IntegerVariable;

/**
 * A VariableNotFoundException is thrown, if an variable is not found. This may
 * happen during statechart conversion if the variable is not defined correct.
 * @author Andre Ohlhoff
 */
public class STRLVariableNotFoundException extends STRLException {
    /**
     * UID.
     */
    private static final long serialVersionUID = 1L;

    /**
     * Constructs a VariableNotFoundException with the given variable.
     * @param v variable, which is not found
     */
    public STRLVariableNotFoundException(final IntegerVariable v) {
        super("variable␣" + v.getName() + "␣not␣found");
    }
}
```

.8 kiel.simulator.syncchart.intexp

.8.1 STRLIntegerAdd.java

```
/*
 * Created on 14.06.2004
 */
package kiel.simulator.syncchart.intexp;

import kiel.simulator.syncchart.exceptions.STRLUninitializedReadException;

/**
 * This is an addition of two integer expressions.
 * @author Andre Ohlhoff
 */
public class STRLIntegerAdd extends STRLIntegerOperator {

    /**
     * Constructs a new addition with the given two sides.
     * @param l
     *            left side
     * @param r
     *            right side
     */
    public STRLIntegerAdd(final STRLIntegerExpression l,
            final STRLIntegerExpression r) {
        super(l, r);
    }

    /**
     * Returns the sum of the left and right value.
     * @return Returns the sum of the left and right value.
     * @throws STRLUninitializedReadException
     *             The exception is thrown if a signal or variable is read
     *             before initialized.
     */
    public final int getValue() throws STRLUninitializedReadException {
        return this.getLeftValue() + this.getRightValue();
    }

}
```

.8.2 STRLIntegerBrackets.java

```java
/*
 * Created on 14.06.2004
 */
package kiel.simulator.syncchart.intexp;

import java.util.Collection;

import kiel.simulator.syncchart.exceptions.STRLUninitializedReadException;

/**
 * This is a bracketed integer expression.
 * @author Andre Ohlhoff
 */
public class STRLIntegerBrackets implements STRLIntegerExpression {

    /**
     * expression which is bracketed.
     */
    private STRLIntegerExpression body = null;

    /**
     * Constructs a bracketed expression.
     * @param b expression to be bracketed.
     */
    public STRLIntegerBrackets(final STRLIntegerExpression b) {
        this.body = b;
    }

    /**
     * Returns the value of the bracketed expression.
     * @return Returns the value of the bracketed expression.
     * @throws STRLUninitializedReadException The exception is thrown if a
     *             signal or variable is read before initialized.
     */
    public final int getValue() throws STRLUninitializedReadException {
        return this.body.getValue();
    }

    /**
     * Returns true, if the bracketed expression can be evaluated.
     * @return Returns true, if the bracketed expression can be evaluated.
     */
    public final boolean ready() {
        return this.body.ready();
    }

    /**
     * Returns the set of all variables read.
     * @return Returns the set of all variables read.
     */
    public final Collection getVariablesUsed() {
        return this.body.getVariablesUsed();
    }
}
```

.8.3 STRLIntegerConstant.java

```java
/*
 * Created on 13.06.2004
 */
package kiel.simulator.syncchart.intexp;

import java.util.Collection;
import java.util.HashSet;

/**
 * This represents a simple integer number.
 * @author Andre Ohlhoff
 */
public class STRLIntegerConstant implements STRLIntegerExpression {

    /**
     * This is the value.
     */
    private int value;

    /**
     * Constructs a new value 0.
     */
    public STRLIntegerConstant() {
        this.value = 0;
    }

    /**
     * Constructs a constant with the given value.
     * @param v value
     */
    public STRLIntegerConstant(final int v) {
        this.setValue(v);
    }

    /**
     * Sets the value.
     * @param v value to be set
     */
    public final void setValue(final int v) {
        this.value = v;
    }

    /**
     * Returns the value.
     * @return Returns the value.
     */
    public final int getValue() {
        return this.value;
    }

    /**
     * Returns true, because the value is always ready.
     * @return Returns true, because the value is always ready.
     */
    public final boolean ready() {
        return true;
    }

    /**
     * Returns an empty set.
     * @return Returns an empty set.
     */
    public final Collection getVariablesUsed() {
        return new HashSet();
    }
}
```

.8.4 STRLIntegerDiv.java

```
/*
 * Created on 14.06.2004
 */
package kiel.simulator.syncchart.intexp;

import kiel.simulator.syncchart.exceptions.STRLUninitializedReadException;

/**
 * This is a division of two integer expressions.
 * @author Andre Ohlhoff
 */
public class STRLIntegerDiv extends STRLIntegerOperator {

    /**
     * Constructs a division.
     * @param l
     *            dividend
     * @param r
     *            divisor
     */
    public STRLIntegerDiv(final STRLIntegerExpression l,
            final STRLIntegerExpression r) {
        super(l, r);
    }

    /**
     * Returns the quotient. Devision by zero is a model error and not handled
     * by the simulator.
     * @return Returns the quotient.
     * @throws STRLUninitializedReadException
     *             The exception is thrown if a signal or variable is read
     *             before initialized.
     */
    public final int getValue() throws STRLUninitializedReadException {
        return this.getLeftValue() / this.getRightValue();
    }

}
```

.8.5 STRLIntegerExpression.java

```
/*
 * Created on 13.06.2004
 */
package kiel.simulator.syncchart.intexp;

import java.util.Collection;

import kiel.simulator.syncchart.exceptions.STRLUninitializedReadException;

/**
 * IntegerExpression is an interface for all kinds of integer expressions.
 * @author Andre Ohlhoff
 */
public interface STRLIntegerExpression {

    /**
     * Evaluates the expression and returns the value.
     * @return Returns the value of the expression.
     * @throws STRLUninitializedReadException
     *             The exception is thrown if a signal or variable is read
     *             before initialized.
     */
    int getValue() throws STRLUninitializedReadException;

    /**
     * Returns true, if the expression can be evaluated. E.g. a combine signal
     * is ready, if it is clear that the signal is not emitted any more in this
     * instant. Using ready does't not prevent for UninitializedReadExceptions
     * which is an error on the statechart, not in the simulator.
     * @return Returns true, if the expression can be evaluated.
     */
    boolean ready();

    /**
     * Returns the set of all variables read.
     * @return Returns the set of all variables read.
     */
    Collection getVariablesUsed();

}
```

.8.6 STRLIntegerMinus.java

```
/*
 * Created on 14.06.2004
 */
package kiel.simulator.syncchart.intexp;

import java.util.Collection;

import kiel.simulator.syncchart.exceptions.STRLUninitializedReadException;

/**
 * This is the unary minus.
 * @author Andre Ohlhoff
 */
public class STRLIntegerMinus implements STRLIntegerExpression {

    /**
     * expression behind the minus.
     */
    private STRLIntegerExpression body = null;

    /**
     * Constructs an unary minus.
     * @param b
     *            expression.
     */
    public STRLIntegerMinus(final STRLIntegerExpression b) {
        this.body = b;
    }

    /**
     * Returns true, if the expressio can be evaluated.
     * @return Returns true, if the expressio can be evaluated.
     */
    public final boolean ready() {
        return this.body.ready();
    }

    /**
     * Returns the value.
     * @return Returns the value.
     * @throws STRLUninitializedReadException
     *             The exception is thrown if a signal or variable is read
     *             before initialized.
     */
    public final int getValue() throws STRLUninitializedReadException {
        return -1 * this.body.getValue();
    }

    /**
     * Returns the set of all variables read.
     * @return Returns the set of all variables read.
     */
    public final Collection getVariablesUsed() {
        return this.body.getVariablesUsed();
    }

}
```

.8.7 STRLIntegerMod.java

```
/*
 * Created on 14.06.2004
 */
package kiel.simulator.syncchart.intexp;

import kiel.simulator.syncchart.exceptions.STRLUninitializedReadException;

/**
 * This is the remainder operator.
 * @author Andre Ohlhoff
 */
public class STRLIntegerMod extends STRLIntegerOperator {

    /**
     * Constructs a remainder operator.
     * @param l
     *            left expression
     * @param r
     *            right expression
     */
    public STRLIntegerMod(final STRLIntegerExpression l,
            final STRLIntegerExpression r) {
        super(l, r);
    }

    /**
     * Returns the remainder.
     * @return Returns the remainder.
     * @throws STRLUninitializedReadException
     *             The exception is thrown if a signal or variable is read
     *             before initialized.
     */
    public final int getValue() throws STRLUninitializedReadException {
        return this.getLeftValue() % this.getRightValue();
    }

}
```

.8.8 STRLIntegerMult.java

```
/*
 * Created on 14.06.2004
 */
package kiel.simulator.syncchart.intexp;

import kiel.simulator.syncchart.exceptions.STRLUninitializedReadException;

/**
 * This is the integer multiplication.
 * @author Andre Ohlhoff
 */
public class STRLIntegerMult extends STRLIntegerOperator {

    /**
     * Constructs a new multiplication.
     * @param l
     *            left factor
     * @param r
     *            right factor
     */
    public STRLIntegerMult(final STRLIntegerExpression l,
            final STRLIntegerExpression r) {
        super(l, r);
    }

    /**
     * Returns the product.
     * @return Returns the product.
     * @throws STRLUninitializedReadException
     *             The exception is thrown if a signal or variable is read
     *             before initialized.
     */
    public final int getValue() throws STRLUninitializedReadException {
        return this.getLeftValue() * this.getRightValue();
    }

}
```

.8.9 STRLIntegerOperator.java

```java
/*
 * Created on 14.06.2004
 */
package kiel.simulator.syncchart.intexp;

import java.util.Collection;
import java.util.HashSet;

import kiel.simulator.syncchart.exceptions.STRLUninitializedReadException;

/**
 * This is the basic class for all integer operators.
 * @author Andre Ohlhoff
 */
public abstract class STRLIntegerOperator implements STRLIntegerExpression {

    /**
     * left parameter.
     */
    private STRLIntegerExpression leftParam = null;

    /**
     * right parameter.
     */
    private STRLIntegerExpression rightParam = null;

    /**
     * Constructs a new operator with left and right parameter.
     * @param l left parameter
     * @param r right parameter
     */
    public STRLIntegerOperator(final STRLIntegerExpression l,
            final STRLIntegerExpression r) {
        this.leftParam = l;
        this.rightParam = r;
    }

    /**
     * Returns the value of the left parameter.
     * @return Returns the value of the left parameter.
     * @throws STRLUninitializedReadException The exception is thrown if a
     *              signal or variable is read before initialized.
     */
    public final int getLeftValue() throws STRLUninitializedReadException {
        return this.leftParam.getValue();
    }

    /**
     * Returns the value of the right parameter.
     * @return Returns the value of the right parameter.
     * @throws STRLUninitializedReadException The exception is thrown if a
     *              signal or variable is read before initialized.
     */
    public final int getRightValue() throws STRLUninitializedReadException {
        return this.rightParam.getValue();
    }

    /**
     * Evaluates the operator. This method is overridden in extending classes.
     * @return Returns the value of the operation.
     * @throws STRLUninitializedReadException The exception is thrown if a
     *              signal or variable is read before initialized.
     */
    public abstract int getValue() throws STRLUninitializedReadException;

    /**
     * Returns true, if both parameters can be evaluated.
     * @return Returns true, if both parameters can be evaluated.
     */
    public final boolean ready() {
        return this.leftParam.ready() && this.rightParam.ready();
    }

    /**
     * Returns the set of all variables read.
     * @return Returns the set of all variables read.
     */
    public final Collection getVariablesUsed() {
        HashSet result = new HashSet();
        result.addAll(this.leftParam.getVariablesUsed());
        result.addAll(this.rightParam.getVariablesUsed());
        return result;
    }
}
```

.8.10 STRLIntegerSub.java

```
/*
 * Created on 14.06.2004
 */
package kiel.simulator.syncchart.intexp;

import kiel.simulator.syncchart.exceptions.STRLUninitializedReadException;

/**
 * This is the integer substraction.
 * @author Andre Ohlhoff
 */
public class STRLIntegerSub extends STRLIntegerOperator {

    /**
     * Constructs a substraction.
     * @param l
     *            minuend
     * @param r
     *            subtrahend
     */
    public STRLIntegerSub(final STRLIntegerExpression l,
            final STRLIntegerExpression r) {
        super(l, r);
    }

    /**
     * Returns the difference.
     * @return Returns the difference.
     * @throws STRLUninitializedReadException
     *             The exception is thrown if a signal or variable is read
     *             before initialized.
     */
    public final int getValue() throws STRLUninitializedReadException {
        return this.getLeftValue() - this.getRightValue();
    }

}
```

.8.11 STRLIntegerVariable.java

```java
/*
 * Created on 13.06.2004
 */
package kiel.simulator.syncchart.intexp;

import java.util.Collection;
import java.util.HashSet;

import kiel.dataStructure.intexp.IntegerVariable;
import kiel.simulator.syncchart.STRLComponent;
import kiel.simulator.syncchart.exceptions.STRLUninitializedReadException;

/**
 * An integer variable.
 * @author Andre Ohlhoff
 */
public class STRLIntegerVariable extends STRLComponent implements
        STRLIntegerExpression {

    /**
     * expression to initialize the variable with.
     */
    private final STRLIntegerExpression initialValue;

    /**
     * is true, if the variable can be initialized with an expression.
     */
    private final boolean mustBeInitialized;

    /**
     * is true, if the variable has a valid value.
     */
    private boolean hasValue = false;

    /**
     * the current value of the variable; only valid if hasValue == true.
     */
    private int currentValue;

    /**
     * reference to the corresponding kiel variable.
     */
    private final IntegerVariable kielVar;

    /**
     * Constructs a integer variable with a reference to the corresponding kiel
     * variable and and integer expression for the initialization.
     * @param iv kiel variable
     * @param v expression for the initialization
     */
    public STRLIntegerVariable(final IntegerVariable iv,
            final STRLIntegerExpression v) {
        this.kielVar = iv;
        this.initialValue = v;
        this.mustBeInitialized = (v != null);
        this.hasValue = false;
    }

    /**
     * Constructs an uninitialized integer variable with a reference to the
     * corresponding kiel variable.
     * @param iv reference to the corresponding kiel variable
     */
    public STRLIntegerVariable(final IntegerVariable iv) {
        this(iv, null);
    }

    /**
     * Returns the expression for initialization.
     * @return Returns the expression for initialization.
     */
    public final STRLIntegerExpression getInitExpression() {
        return this.initialValue;
    }

    /**
     * Returns the value of the variable.
     * @return Returns the value of the variable.
     * @throws STRLUninitializedReadException The exception is thrown if the
     *              variable is read before initialized.
     */
    public final int getValue() throws STRLUninitializedReadException {
        if (this.hasValue) {
            return this.currentValue;
        } else {
            throw new STRLUninitializedReadException(this.getKielVar());
        }
    }

    /**
     * Returns true, if the variable has a valid value.
     * @return Returns true, if the variable has a valid value.
     */
    public final boolean hasValue() {
        return this.hasValue;
    }

    /**
     * Sets the value.
     * @param value value to be set
     */
    public final void setValue(final int value) {
        this.currentValue = value;
        this.hasValue = true;
        this.getContext().logVarValue(this.getKielVar(), new Integer(value));
    }

    /**
     * Initializes the variable.
     * @throws STRLUninitializedReadException The exception is thrown if a
     *              signal or variable is read before initialized.
     */
    public final void initializeValue() throws STRLUninitializedReadException {
        if (this.mustBeInitialized) {
            this.setValue(this.initialValue.getValue());
        } else {
            this.hasValue = false;
            this.getContext().logVarValue(this.getKielVar(), null);
        }
    }

    /**
     * Returns always true. Variables can be read at any time.
     * @return Returns always true.
     */
    public final boolean ready() {
        return true;
    }

    /**
     * Returns a reference to the corresponding kiel variable.
     * @return Returns a reference to the corresponding kiel variable.
     */
    public final IntegerVariable getKielVar() {
        return this.kielVar;
    }

    /**
     * Returns a set containing this variable.
     * @return Returns a set containing this variable.
     */
    public final Collection getVariablesUsed() {
        HashSet result = new HashSet();
        result.add(this);
        return result;
    }

    /**
     * Returns true, if the variable must be initialized.
     * @return Returns true, if the variable must be initialized.
     */
    public final boolean mustBeInitialized() {
        return this.mustBeInitialized;
    }
}
```

.8.12 STRLVariableHandler.java

```
/*
 * Created on 22.09.2004
 */
package kiel.simulator.syncchart.intexp;

import java.util.LinkedList;

import kiel.simulator.syncchart.STRLComponent;
import kiel.simulator.syncchart.STRLSimulator;
import kiel.simulator.syncchart.action.STRLAction;
import kiel.simulator.syncchart.action.STRLActions;
import kiel.simulator.syncchart.action.STRLInitializeVariable;
import kiel.simulator.syncchart.exceptions.STRLException;

/**
 * A VariableHandler takes care of a list of variables which is defined in one
 * stg / region.
 * @author Andre Ohlhoff
 */
public class STRLVariableHandler extends STRLComponent {

    /**
     * list of variables.
     */
    private LinkedList variables = new LinkedList();

    /**
     * Actions used to initialize the variables.
     */
    private STRLActions initActions = null;

    /**
     * Sets the variables.
     * @param vars variables to be set
     */
    public final void setVariables(final STRLIntegerVariable[] vars) {
        STRLAction[] actions = new STRLAction[vars.length];
        for (int i = 0; i < vars.length; i++) {
            this.variables.add(vars[i]);
            actions[i] = new STRLInitializeVariable(vars[i]);
        }
        this.initActions = new STRLActions(actions);
    }

    /**
     * Resets all variables.
     * @throws STRLException The exception is thrown if a signal or variable is
     *             read before initialized.
     */
    public final void initialize() throws STRLException {
        this.initActions.execute();
    }

    /**
     * Sets the context for all variables.
     * @param sim context
     */
    public final void setComponentsContext(final STRLSimulator sim) {
        STRLComponent.setContext(sim, this.variables);
        this.initActions.setContext(sim);
    }
}
```

.9 kiel.simulator.syncchart.sigexp

.9.1 STRLCombineSignal.java

```java
/*
 * Created on 08.09.2004
 */
package kiel.simulator.syncchart.sigexp;

import java.util.Collection;

import kiel.dataStructure.eventexp.CombineSignal;
import kiel.dataStructure.eventexp.IntegerSignal;
import kiel.simulator.syncchart.exceptions.STRLUninitializedReadException;
import kiel.simulator.syncchart.intexp.STRLIntegerExpression;

/**
 * A combine signal can be emitted arbitrary times a instant. All values are
 * combined with a commutative and associative combining function.
 * @author Andre Ohlhoff
 */
public abstract class STRLCombineSignal extends STRLIntegerSignal {

    /**
     * is true, if the signal is already emitted this instant.
     */
    private boolean emittedThisInstant = false;

    /**
     * Constructs a new combine signal with a reference to the corresponding
     * kiel combine signal and an expression for initialization.
     * @param cs kiel combine signal
     * @param value expression for initialization
     */
    public STRLCombineSignal(final CombineSignal cs,
            final STRLIntegerExpression value) {
        super(cs, value);
    }

    /**
     * Constructs an uninitialized combine signal with a reference to the
     * corresponding kiel combine signal.
     * @param cs kiel combine signal
     */
    public STRLCombineSignal(final CombineSignal cs) {
        super(cs);
    }

    /**
     * Emits the signal with the given value.
     * @param value value
     */
    public final void emitValue(final int value) {
        if (this.emittedThisInstant) {
            try {
                this.setCurrentValue(this.intComb(this.getValue(), value));
            } catch (STRLUninitializedReadException e) {
                // dead code
                this.getContext().debug("ohoh");
            }
        } else {
            this.emittedThisInstant = true;

            this.setCurrentValue(value);
        }
    }

    /**
     * Sets the current status to present if the current status is unknown.
     */
    public final void emit() {
        if (this.getCurrentStatus() == STRLSignal.UNKNOWN) {
            this.setCurrentStatus(STRLSignal.PRESENT);
        }
    }

    /**
     * Initializes the signal status.
     */
    public final void initializeStatus() {
        this.setReady(false);
        this.emittedThisInstant = false;
        this.setCurrentStatus(STRLSignal.UNKNOWN);
        this.setPreviousStatus(STRLSignal.UNKNOWN);
    }

    /**
     * Prepares the signal for the next instant.
     */
    public final void resetStatus() {
        this.setReady(false);
        this.emittedThisInstant = false;
        if (this.hasValue()) {
            try {
                this.setPreviousValue(this.getValue());
            } catch (STRLUninitializedReadException e) {
                // dead code
                this.getContext().debug("ohoh");
            }
        }
        this.setPreviousStatus(this.getCurrentStatus());
        this.setCurrentStatus(STRLSignal.UNKNOWN);
    }

    /**
     * Sets the current status to absent, if the signal is not contained by the
     * given collection and the current state if still unknown.
     * @param signals Set of potentially emitted signals.
     */
    public final void update(final Collection signals) {
        if (!this.ready() && !signals.contains(this)) {
            if (!this.emittedThisInstant) {
                this.setCurrentStatus(STRLSignal.ABSENT);
            }
            if (this.hasValue()) {

                Integer value;
                try {
                    value = new Integer(this.getValue());
                    this.getContext().logSigValue(
                            (IntegerSignal) this.getKielEvent(), value);
                } catch (STRLUninitializedReadException e) {
                    // dead code
                    this.getContext().debug("ohoh");
                }
            }

            this.setReady(true);
        }
    }

    /**
     * Commutative and associative combining function which must be implemented
     * in extending classes.
     * @param a first value
     * @param b second value
     * @return combined value
     */
    public abstract int intComb(final int a, final int b);
}
```

.9.2 STRLCombineWithAdd.java

```
/*
 * Created on 08.09.2004
 */
package kiel.simulator.syncchart.sigexp;

import kiel.dataStructure.eventexp.CombineWithAdd;
import kiel.simulator.syncchart.intexp.STRLIntegerExpression;

/**
 * A combine with add signal uses addition as combining function.
 * @author Andre Ohlhoff
 */
public class STRLCombineWithAdd extends STRLCombineSignal {

    /**
     * Constructs a new combine with add signal.
     * @param cwa
     *            kiel combine with add
     * @param value
     *            expression for initialization
     */
    public STRLCombineWithAdd(final CombineWithAdd cwa,
            final STRLIntegerExpression value) {
        super(cwa, value);
    }

    /**
     * Constructs an uninitialized combine with add signal.
     * @param cwa
     *            kiel combine with add
     */
    public STRLCombineWithAdd(final CombineWithAdd cwa) {
        super(cwa);
    }

    /**
     * Addition as combining function.
     * @param a
     *            first addend
     * @param b
     *            second addend
     * @return sum
     */
    public final int intComb(final int a, final int b) {
        return a + b;
    }
}
```

.9.3 STRLCombineWithMult.java

```
/*
 * Created on 08.09.2004
 */
package kiel.simulator.syncchart.sigexp;

import kiel.dataStructure.eventexp.CombineWithMult;
import kiel.simulator.syncchart.intexp.STRLIntegerExpression;

/**
 * A combine with mult signal uses muliplication as combining function.
 * @author Andre Ohlhoff
 */
public class STRLCombineWithMult extends STRLCombineSignal {

    /**
     * Constructs a combine with mult signal.
     * @param cwm
     *            kiel combine with mult
     * @param value
     *            expression for initialization.
     */
    public STRLCombineWithMult(final CombineWithMult cwm,
            final STRLIntegerExpression value) {
        super(cwm, value);
    }

    /**
     * Constructs an uninitialized combine with mult signal.
     * @param cwm
     *            kiel combine with mult.
     */
    public STRLCombineWithMult(final CombineWithMult cwm) {
        super(cwm);
    }

    /**
     * Multiplication as combining function.
     * @param a
     *            first factor
     * @param b
     *            second factor
     * @return product
     */
    public final int intComb(final int a, final int b) {
        return a * b;
    }

}
```

.9.4 STRLIntegerSignal.java

```java
/*
 * Created on 13.06.2004
 */
package kiel.simulator.syncchart.sigexp;

import java.util.Collection;
import java.util.HashSet;

import kiel.dataStructure.eventexp.IntegerSignal;
import kiel.simulator.syncchart.exceptions.STRLException;
import kiel.simulator.syncchart.exceptions.STRLUninitializedReadException;
import kiel.simulator.syncchart.intexp.STRLIntegerExpression;

/**
 * An integer signal has a status and a value.
 * @author Andre Ohlhoff
 */
public abstract class STRLIntegerSignal extends STRLSignal implements
        STRLIntegerExpression {

    /**
     * is true, if the signal is ready.
     */
    private boolean isReady = false;

    /**
     * is true, if the value of the signal must be initialized.
     */
    private final boolean mustBeInitialized;

    /**
     * is true, if the signal has a valid value.
     */
    private boolean hasValue = false;

    /**
     * is true, if the signal had a valid value in the previous instant.
     */
    private boolean hasPreviousValue = false;

    /**
     * integer expression for initialization.
     */
    private final STRLIntegerExpression initialValue;

    /**
     * the current value of the signal is only valid if hasValue == true.
     */
    private int currentValue = 0;

    /**
     * the previous value of the signal is only valid if hasPreviousValue ==
     * true.
     */
    private int previousValue = 0;

/**
 * Constructs an integer signal with a reference to the corresponding kiel
 * integer signal and an expression for initialization.
 * @param is kiel integer signal
 * @param value expression for initialization
 */
public STRLIntegerSignal(final IntegerSignal is,
        final STRLIntegerExpression value) {
    super(is);
    this.initialValue = value;
    this.mustBeInitialized = (value != null);
    this.hasValue = false;
}

/**
 * Constructs an uninitialized integer signal with a reference to the
 * corresponding kiel integer signal.
 * @param is kiel integer signal
 */
public STRLIntegerSignal(final IntegerSignal is) {
    this(is, null);
}

/**
 * Returns true, if the signal must be initialized.
 * @return Returns true, if the signal must be initialized.
 */
public final boolean mustBeInitialized() {
    return this.mustBeInitialized;
}

/**
 * Returns the expression for the initialization.
 * @return Returns the expression for the initialization.
 */
public final STRLIntegerExpression getInitExpression() {
    return this.initialValue;
}

/**
 * Returns true, if the signal has a valid value.
 * @return Returns true, if the signal has a valid value.
 */
public final boolean hasValue() {
    return this.hasValue;
}

/**
 * Returns the current value.
 * @return Returns the current value.der aktuelle Wert.
 * @throws STRLUninitializedReadException The exception is thrown, if the
 *              signal has no valid value.
 */
public final int getValue() throws STRLUninitializedReadException {
    if (this.hasValue) {
        return this.currentValue;
        } else {
            throw new STRLUninitializedReadException((IntegerSignal) this
                    .getKielEvent());
        }
    }

    /**
     * Returns the previous value.
     * @return Returns the previous value.
     * @throws STRLUninitializedReadException the exception is thrown if the
     *             signal has no previous value.
     */
    public final int getPreviousValue() throws STRLUninitializedReadException {
        if (this.hasPreviousValue) {
            return this.previousValue;
        } else {
            throw new STRLUninitializedReadException("signal␣"
                    + this.getKielEvent().getName() + "␣has␣no␣previous␣value")
     ;
        }
    }

    /**
     * Sets the current value.
     * @param value new current value
     */
    public final void setCurrentValue(final int value) {
        this.currentValue = value;
        this.hasValue = true;
    }

    /**
     * Sets the previous value.
     * @param value new previous value
     */
    public final void setPreviousValue(final int value) {
        this.previousValue = value;
        this.hasPreviousValue = true;
    }

    /**
     * Returns true, if the signal is ready.
     * @return Returns true, if the signal is ready.
     */
    public final boolean ready() {
        return this.isReady;
    }

    /**
     * Sets the isReady flag.
     * @param ready true, if this signal is ready.
     */
    public final void setReady(final boolean ready) {
        this.isReady = ready;
    }

    /**
     * Initializes the signal value.
     * @throws STRLUninitializedReadException The exception is thrown if a
     *             signal or variable is read before initialized.
     */
    public final void initializeValue() throws STRLUninitializedReadException {
        if (this.mustBeInitialized()) {
            this.setCurrentValue(this.getInitExpression().getValue());
            this.setPreviousValue(this.getValue());
        } else {
            this.hasValue = false;
            this.hasPreviousValue = false;
        }
        this.getContext().debug(
                "initialized␣signal␣" + this.getKielEvent().getName());
    }

    /**
     * Sets the current state to present if the current state if unknown. Emit
     * is used together with emitValue.
     * @throws STRLException The Exception is thrown, if a single signal is
     *             emitted twice.
     */
    public abstract void emit() throws STRLException;

    /**
     * Emits the value of the signal.
     * @param value value
     */
    public abstract void emitValue(final int value);

    /**
     * Returns an empty set.
     * @return Returns an empty set.
     */
    public final Collection getVariablesUsed() {
        return new HashSet();
    }

}
```

.9.5 STRLPre.java

```
/*
 * Created on 29.09.2004
 */
package kiel.simulator.syncchart.sigexp;

import java.util.Collection;
import java.util.HashSet;

import kiel.simulator.syncchart.boolexp.STRLBooleanExpression;
import kiel.simulator.syncchart.exceptions.STRLUninitializedReadException;
import kiel.simulator.syncchart.intexp.STRLIntegerExpression;

/**
 * The pre expression is used in signal expressions (await a and pre(b)) or in
 * integer expressions (v := 2 + pre(?s)).
 * @author Andre Ohlhoff
 */
public class STRLPre implements STRLBooleanExpression, STRLIntegerExpression {

    /**
     * the reference to a boolean expression is not null, if pre is used as a
     * signal expression.
     */
    private STRLBooleanExpression sigExp = null;

    /**
     * the reference to an integer signals is not null, if pre is used as an
     * integer expression.
     */
    private STRLIntegerSignal intSig = null;

    /**
     * Sets the signal expression.
     * @param exp
     *            signal expression
     */
    public final void setExpression(final STRLBooleanExpression exp) {
        this.sigExp = exp;
    }

    /**
     * Sets the integer signal.
     * @param signal
     *            integer signal
     */
    public final void setIntSig(final STRLIntegerSignal signal) {
        this.intSig = signal;
    }

    /**
     * Evaluates the pre as a signal expression.
     * @param fresh
     *            fresh signals
     * @param unknown
     *            unknown signals
     * @return Wert des Ausdrucks.
 * @see STRLBooleanExpression#evaluate(Collection, Collection)
 */
public final int evaluate(final Collection fresh,
        final Collection unknown) {
    return this.sigExp.pre0(fresh, unknown);
}

/**
 * Returns the value of the contained integer signal in the previous
 * instant.
 * @return Returns the value of the contained integer signal in the previous
 *         instant.
 * @throws STRLUninitializedReadException
 *             The excetpion is thrown if the signal has no valid previous
 *             value.
 */
public final int getValue() throws STRLUninitializedReadException {
    return this.intSig.getPreviousValue();
}

/**
 * Returns true.
 * @return Returns true.
 */
public final boolean ready() {
    return true;
}

/**
 * Nested pre is not supported.
 * @param fresh
 *            ignored
 * @param unknown
 *            ignored
 * @return -1
 * @see STRLBooleanExpression#pre0(Collection, Collection)
 */
public final int pre0(final Collection fresh, final Collection unknown) {
    return -1; // dead code
}

/**
 * Nested pre is not supported.
 * @param fresh
 *            ignored
 * @param unknown
 *            ignored
 * @return -1
 * @see STRLBooleanExpression#pre1(Collection, Collection)
 */
public final int pre1(final Collection fresh, final Collection unknown) {
    return -1; // dead code
}

/**
 * Returns an empty set.
 * @return Returns an empty set.
 */
public final Collection getVariablesUsed() {
    return new HashSet();
    }
}
```

.9.6 STRLPureSignal.java

```java
/*
 * Created on 09.10.2004
 */
package kiel.simulator.syncchart.sigexp;

import java.util.Collection;
import java.util.HashSet;

import kiel.dataStructure.eventexp.Signal;

/**
 * A pure signal has only a status and no value.
 * @author Andre Ohlhoff
 */
public class STRLPureSignal extends STRLSignal {

    /**
     * Constructs a pure signal with a reference to the corresenponding kiel
     * signal.
     * @param s
     *            kiel signal
     */
    public STRLPureSignal(final Signal s) {
        super(s);
    }

    /**
     * Initializes the signal signal status.
     */
    public final void initializeStatus() {
        this.setCurrentStatus(STRLSignal.UNKNOWN);
        this.setPreviousStatus(STRLSignal.UNKNOWN);
    }

    /**
     * Nothing happens because a pure signal has no value.
     */
    public final void initializeValue() {
        this.getContext().debug(
                "initialized␣signal␣" + this.getKielEvent().getName());
    }

    /**
     * Prepares the pure signal for the next instant. The previous status is set
     * to the current status und the current status is set to unknown.
     */
    public final void resetStatus() {
        this.setPreviousStatus(this.getCurrentStatus());
        this.setCurrentStatus(STRLSignal.UNKNOWN);
    }

    /**
     * Emits the signal. If the current state is unknown, it is set to present.
     */
    public final void emit() {
        if (this.getCurrentStatus() == STRLSignal.UNKNOWN) {
            this.setCurrentStatus(STRLSignal.PRESENT);
        }
    }

    /**
     * Sets the current state to absent, if the signal is not contained by the
     * give collection and the current state is still unknown.
     * @param signals
     *            set of potentially emittes signals.
     */
    public final void update(final Collection signals) {
        if (this.getCurrentStatus() == STRLSignal.UNKNOWN
                && !signals.contains(this)) {
            this.setCurrentStatus(STRLSignal.ABSENT);
        }
    }

    /**
     * Returns the collection of all used variables.
     * @return Returns the collection of all used variables.
     */
    public final Collection getVariablesUsed() {
        return new HashSet();
    }
}
```

.9.7 STRLSignal.java

```java
/*
 * Created on 14.05.2004
 */
package kiel.simulator.syncchart.sigexp;

import java.util.Collection;

import kiel.dataStructure.eventexp.Event;
import kiel.dataStructure.eventexp.Signal;
import kiel.simulator.syncchart.STRLComponent;
import kiel.simulator.syncchart.boolexp.STRLBooleanExpression;
import kiel.simulator.syncchart.exceptions.STRLUninitializedReadException;

/**
 * Signal is the basic class for all signal types.
 * @author Andre Ohlhoff
 */
public abstract class STRLSignal extends STRLComponent implements
        STRLBooleanExpression {

    /**
     * value for an absent signal.
     */
    public static final int ABSENT = STRLBooleanExpression.FALSE;

    /**
     * value for a present signal.
     */
    public static final int PRESENT = STRLBooleanExpression.TRUE;

    /**
     * current status of the signal.
     */
    private int currentStatus = STRLSignal.UNKNOWN;

    /**
     * previous status of the signal.
     */
    private int previousStatus = STRLSignal.UNKNOWN;

    /**
     * reference to the corresponding kiel event.
     */
    private Event kielEvent = null;

    /**
     * Constructs a signal with a reference to the corresponding kiel event.
     * @param ke kiel event
     */
    public STRLSignal(final Event ke) {
        this.kielEvent = ke;
    }

    /**
     * Returns a reference to the corresponding kiel event.
     * @return Returns a reference to the corresponding kiel event.
     */
    public final Event getKielEvent() {
        return this.kielEvent;
    }

    /**
     * Returns the current status of the signal. If one of the given collections
     * constains the signal unknown is returned instead of the real status.
     * @param fresh fresh signals
     * @param unknown unknown signals
     * @return Returns the current status of the signal.
     * @see STRLBooleanExpression#evaluate(Collection, Collection)
     */
    public final int evaluate(final Collection fresh,
            final Collection unknown) {
        int result;
        if (fresh.contains(this) || unknown.contains(this)) {
            result = STRLSignal.UNKNOWN;
        } else {
            result = this.currentStatus;
        }
        return result;
    }

    /**
     * Sets the current status.
     * @param status the new current status
     */
    public final void setCurrentStatus(final int status) {
        if (this.getContext() != null) {
            if (status == STRLSignal.PRESENT) {
                this.getContext().logPresent((Signal) this.kielEvent);
            } else if (status == STRLSignal.ABSENT) {
                this.getContext().logAbsent((Signal) this.getKielEvent());
            } else {
                this.getContext().logUnknown((Signal) this.getKielEvent());
            }
        }
        this.currentStatus = status;
    }

    /**
     * Sets the previous status.
     * @param prevStatus the new previous status
     */
    public final void setPreviousStatus(final int prevStatus) {
        this.previousStatus = prevStatus;
    }

    /**
     * Returns the current status.
     * @return Returns the current status.
     */
    public final int getCurrentStatus() {
        return this.currentStatus;
    }
    /**
     * Returns the previous status of the signal. If the collection fresh
     * contains the signal, absent is returned. If the collection unknown
     * contains the signal, the current status is returned. If the previous
     * status is unknown, absent is returned.
     * @param fresh fresh signals
     * @param unknown unknown signals
     * @return Returns the previous status of the signal.
     * @see STRLBooleanExpression#pre0(Collection, Collection)
     */
    public final int pre0(final Collection fresh, final Collection unknown) {
        int result;
        if (fresh.contains(this)) {
            result = STRLSignal.ABSENT;
        } else if (unknown.contains(this)) {
            result = this.currentStatus;
        } else {
            result = this.previousStatus;
        }
        if (result == STRLSignal.UNKNOWN) {
            result = STRLSignal.ABSENT;
        }
        return result;
    }

    /**
     * Returns the previous status of the signal. If the collection fresh
     * contains the signal, present is returned. If the collection unknown
     * contains the signal, the current status is returned. If the previous
     * status is unknown, present is returned.
     * @param fresh fresh signals
     * @param unknown unknown signals
     * @return Returns the previous status of the signal.
     * @see STRLBooleanExpression#pre1(Collection, Collection)
     */
    public final int pre1(final Collection fresh, final Collection unknown) {
        int result;
        if (fresh.contains(this)) {
            result = STRLSignal.PRESENT;
        } else if (unknown.contains(this)) {
            result = this.currentStatus;
        } else {
            result = this.previousStatus;
        }
        if (result == STRLSignal.UNKNOWN) {
            result = STRLSignal.PRESENT;
        }
        return result;
    }

    /**
     * Prepares a signal for the next instant, if the signal is visible in this
     * instant.
     */
    public abstract void resetStatus();

    /**
     * Initializes the value of the signal.
     * @throws STRLUninitializedReadException The exception is thrown if a
     *             signal or variable is read before initialized.
     */
    public abstract void initializeValue()
            throws STRLUninitializedReadException;

    /**
     * Intializes the status of the signal.
     */
    public abstract void initializeStatus();

    /**
     * Sets the current state to absent, if the signal is not contained in the
     * given collection and the current state is still unknown.
     * @param potential set of potentially emitted signals
     */
    public abstract void update(final Collection potential);
}
```

.9.8 STRLSignalHandler.java

```
/*
 * Created on 17.09.2004
 */
package kiel.simulator.syncchart.sigexp;

import java.util.Collection;
import java.util.Iterator;
import java.util.LinkedList;

import kiel.dataStructure.eventexp.IntegerSignal;
import kiel.dataStructure.eventexp.Signal;
import kiel.simulator.syncchart.STRLComponent;
import kiel.simulator.syncchart.STRLSimulator;
import kiel.simulator.syncchart.action.STRLAction;
import kiel.simulator.syncchart.action.STRLActions;
import kiel.simulator.syncchart.action.STRLInitializeSignal;
import kiel.simulator.syncchart.exceptions.STRLException;
import kiel.simulator.syncchart.exceptions.STRLIntegerSignalNotFoundException;
import kiel.simulator.syncchart.exceptions.STRLSignalNotFoundException;

/**
 * A SignalHandler takes care of a set of signals.
 * @author Andre Ohlhoff
 */
public class STRLSignalHandler extends STRLComponent {

    /**
     * list of signals.
     */
    private LinkedList signals = new LinkedList();

    /**
     * actions used to initialize the signals.
     */
    private STRLActions initActions = null;

    /**
     * Sets the signals.
     * @param s array of signals
     */
    public final void setSignals(final STRLSignal[] s) {
        STRLAction[] actions = new STRLAction[s.length];
        for (int i = 0; i < s.length; i++) {
            this.signals.addLast(s[i]);
            actions[i] = new STRLInitializeSignal(s[i]);
        }
        this.initActions = new STRLActions(actions);
    }

    /**
     * Sets the context for the signals.
     * @param c Der Context fÃ¼r die Signale.
     */
    public final void setComponentsContext(final STRLSimulator c) {
        STRLComponent.setContext(c, this.signals);
        this.initActions.setContext(c);
    }

    /**
     * Updates the signals.
     * @param potential set of potentially emittes signals
     */
    public final void update(final Collection potential) {
        Iterator signalIter = this.signals.iterator();
        while (signalIter.hasNext()) {
            ((STRLSignal) signalIter.next()).update(potential);
        }
    }

    /**
     * Initializes the signals.
     * @throws STRLException The exception is thrown if a signal or variable is
     *              read before initialized.
     */
    public final void initialize() throws STRLException {
        this.initActions.execute();
    }

    /**
     * Resets the status of the signals, so that they are ready to be used in
     * the next instant.
     */
    public final void resetStatus() {
        Iterator signalIter = this.signals.iterator();
        while (signalIter.hasNext()) {
            ((STRLSignal) signalIter.next()).resetStatus();
        }
    }

    /**
     * Emits the corresponding syncchart signal.
     * @param signal kiel signal
     * @throws STRLException The exception is thrown, if this signal is not
     *              found, or if the signals must be emitted with a value.
     */
    public final void emit(final Signal signal) throws STRLException {
        Iterator signalIter = this.signals.iterator();
        STRLSignal s = null;
        while (signalIter.hasNext()) {
            s = (STRLSignal) signalIter.next();
            if (s.getKielEvent().equals(signal)) {
                break;
            }
        }
        if (s != null) {
            if (s instanceof STRLPureSignal) {
                ((STRLPureSignal) s).emit();
            } else {
                throw new STRLException("cannot␣emit␣" + signal.getName()
                        + "␣without␣a␣value");
            }
        } else {
            throw new STRLSignalNotFoundException(signal);
        }
    }

    /**
     * Emits this corresponding syncchart signal with the given value.
     * @param signal kiel signal
     * @param value value to emit
     * @throws STRLException The exception is thrown if the signal is not found,
     *              or if the signal cannot be emitted with a value.
     */
    public final void emit(final IntegerSignal signal, final int value)
            throws STRLException {
        Iterator signalIter = this.signals.iterator();
        STRLSignal s = null;
        while (signalIter.hasNext()) {
            s = (STRLSignal) signalIter.next();
            if (s.getKielEvent().equals(signal)) {
                break;
            }
        }
        if (s != null) {
            if (s instanceof STRLIntegerSignal) {
                ((STRLIntegerSignal) s).emit();
                ((STRLIntegerSignal) s).emitValue(value);
            }
        } else {
            throw new STRLIntegerSignalNotFoundException(signal);
        }
    }

    /**
     * Returns the signals.
     * @return Returns the signals.
     */
    public final Collection getSignals() {
        return this.signals;
    }
}
```

.9.9 STRLSignalTick.java

```
package kiel.simulator.syncchart.sigexp;

import java.util.Collection;
import java.util.HashSet;

import kiel.dataStructure.eventexp.Tick;

/**
 * The signal tick is a prefined signal present in every instant.
 * @author Andre Ohlhoff
 */
public class STRLSignalTick extends STRLSignal {

    /**
     * Constructs tick.
     * @param t kiel tick
     */
    public STRLSignalTick(final Tick t) {
        super(t);
        this.setCurrentStatus(STRLSignal.PRESENT);
    }

    /**
     * does nothing.
     */
    public final void resetStatus() {
    }

    /**
     * does nothing.
     */
    public final void initializeValue() {
    }

    /**
     * does nothing.
     */
    public final void initializeStatus() {
    }

    /**
     * does nothing.
     * @param potential ignored
     */
    public final void update(final Collection potential) {
    }

    /**
     * Returns the collection of all used variables.
     * @return Returns the collection of all used variables.
     */
    public final Collection getVariablesUsed() {
        return new HashSet();
    }

}
```

.9.10 STRLSignalTrue.java

```java
/*
 * Created on 28.05.2004
 */
package kiel.simulator.syncchart.sigexp;

import kiel.dataStructure.eventexp.TrueSignal;

/**
 * SignalTrue is a always true dummy signal, which is used as default trigger in
 * transitions without a trigger.
 * @author Andre Ohlhoff
 */
public class STRLSignalTrue extends STRLSignalTick {

    /**
     * Construcst a true signal.
     * @param ts
     *            kiel true signal
     */
    public STRLSignalTrue(final TrueSignal ts) {
        super(ts);
    }
}
```

.9.11 STRLSingleSignal.java

```
/*
 * Created on 09.10.2004
 */
package kiel.simulator.syncchart.sigexp;

import java.util.Collection;

import kiel.dataStructure.eventexp.IntegerSignal;
import kiel.simulator.syncchart.exceptions.STRLException;
import kiel.simulator.syncchart.exceptions.STRLUninitializedReadException;
import kiel.simulator.syncchart.intexp.STRLIntegerExpression;

/**
 * A single signal can only be emitted once a instant.
 * @author Andre Ohlhoff
 */
public class STRLSingleSignal extends STRLIntegerSignal {

    /**
     * Constructs a single signal with a reference to the corresponding kiel
     * integer expression and an expression for initialization.
     * @param is kiel integer signal
     * @param value expression for initialization
     */
    public STRLSingleSignal(final IntegerSignal is,
            final STRLIntegerExpression value) {
        super(is, value);
    }

    /**
     * Constructs an uninitialized single signal with a reference to the
     * corresponding kiel integer signal.
     * @param is Kiel IntegerSignal
     */
    public STRLSingleSignal(final IntegerSignal is) {
        super(is);
    }

    /**
     * Initializes the signal status.
     */
    public final void initializeStatus() {
        this.setCurrentStatus(STRLSignal.UNKNOWN);
        this.setPreviousStatus(STRLSignal.UNKNOWN);
        this.setReady(false);
    }

    /**
     * Prepares the signal for the next instant.
     */
    public final void resetStatus() {
        if (this.hasValue()) {
            try {
                this.setPreviousValue(this.getValue());
            } catch (STRLUninitializedReadException e) {
                // dead code
                this.getContext().debug("ohoh");
            }
        }
        this.setReady(false);
        this.setPreviousStatus(this.getCurrentStatus());
        this.setCurrentStatus(STRLSignal.UNKNOWN);
    }

    /**
     * Sets the current value and sets the isReady flag to true.
     * @param value new value
     */
    public final void emitValue(final int value) {
        this.getContext().logSigValue((IntegerSignal) this.getKielEvent(),
                new Integer(value));
        this.setCurrentValue(value);
        this.setReady(true);
    }

    /**
     * Emits the signal.
     * @throws STRLException The exception is thrown, if the status of the
     *             signal is already present.
     */
    public final void emit() throws STRLException {
        if (this.getCurrentStatus() == STRLSignal.PRESENT) {
            throw new STRLException("single␣signal:␣"
                    + this.getKielEvent().getName() + "␣emitted␣twice.");
        } else if (this.getCurrentStatus() == STRLSignal.UNKNOWN) {
            this.setCurrentStatus(STRLSignal.PRESENT);
        }

    }

    /**
     * Sets the current status to absent, if the signal is not contained by the
     * given collection and the current state is still unknown.
     * @param signals Set of potentially emitted signals.
     */
    public final void update(final Collection signals) {
        if (!this.ready() && !signals.contains(this)) {
            if (this.getCurrentStatus() == STRLSignal.UNKNOWN) {
                this.setCurrentStatus(STRLSignal.ABSENT);
                Integer value = null;
                if (this.hasValue()) {
                    try {
                        value = new Integer(this.getValue());
                    } catch (STRLUninitializedReadException e) {
                        // dead code
                        this.getContext().debug("ohoh");
                    }
                }
                this.getContext().logSigValue(
                        (IntegerSignal) this.getKielEvent(), value);
                this.setReady(true);
            }
        }
    }
}
```

.10 kiel.simulator.syncchart.transition

.10.1 STRLDelayExpression.java

```java
/*
 * Created on 14.09.2004
 */
package kiel.simulator.syncchart.transition;

import java.util.Collection;

import kiel.dataStructure.eventexp.TrueSignal;
import kiel.simulator.syncchart.boolexp.STRLBooleanExpression;
import kiel.simulator.syncchart.exceptions.STRLException;
import kiel.simulator.syncchart.exceptions.STRLUninitializedReadException;
import kiel.simulator.syncchart.intexp.STRLIntegerConstant;
import kiel.simulator.syncchart.intexp.STRLIntegerExpression;
import kiel.simulator.syncchart.sigexp.STRLSignalTrue;

/**
 * A DelayExpression consists of a signal expression (boolean expression over
 * signal statuses), an immediate flag and an integer expression as count delay.
 * The immediate flag should only be set to true, if the count delay is set to
 * default.
 * @author Andre Ohlhoff
 */
public class STRLDelayExpression {

    /**
     * This constant is used as return value, showing that the delayExpression
     * is not satisfied.
     */
    public static final int NOTSATISFIED = 0;

    /**
     * This constant is used as return value, showing that the delayExpression
     * is satiesfied.
     */
    public static final int SATISFIED = 1;

    /**
     * This constant is used as return value, showing that is unknown whether
     * the delayExpression is satiesfied or not.
     */
    public static final int UNKNOWN = 2;

    /**
     * If this flag is fals, the delayExpression cannot be satisfied in the
     * first instant of the corresponding cell.
     */
    private boolean immediateFlag = false;

    /**
     * Storing the return value of 'evaluate' eleminates redundant computations.
     */
    private int lastResult = STRLDelayExpression.UNKNOWN;

    /**
     * The count delay specifies how many instants signal expression must be
     * satiesfied, before the delay expression is satisfied.
     */
    private STRLIntegerExpression countDelay = new STRLIntegerConstant(1);

    /**
     * Signal expression which is evaluated.
     */
    private STRLBooleanExpression signalExp = new STRLSignalTrue(
            new TrueSignal());

    /**
     * Stores the value of the count delay expression in the moment when the
     * corresponding cell is entered.
     */
    private int count = 1;

    /**
     * Counts from 'count' to zero.
     */
    private int left = 1;

    /**
     * Sets the immediate flag to true.
     */
    public final void setImmediateFlag() {
        this.immediateFlag = true;
    }

    /**
     * Sets the count delay integer expression.
     * @param intExp
     *            count delay integer expression.
     */
    public final void setCountDelay(final STRLIntegerExpression intExp) {
        this.countDelay = intExp;
    }

    /**
     * Returns the count delay integer expression.
     * @return Returns the count delay integer expression.
     */
    public final STRLIntegerExpression getCountDelay() {
        return this.countDelay;
    }

    /**
     * Sets the signal expression.
     * @param sigExp
     *            signal expression.
     */
    public final void setSignalExp(final STRLBooleanExpression sigExp) {
        this.signalExp = sigExp;
    }

    /**
     * Resets the stored 'lastResult' and sets the value of 'left' to the value
     * of 'count'.
     */
    public final void prepareForNext() {
        this.lastResult = STRLDelayExpression.UNKNOWN;
        if (this.left == 0) {
            this.left = this.count;
        }
    }

    /**
     * Initializes the delay expression. Evaluates the count delay and stores
     * its value in 'count'.
     * @throws STRLUninitializedReadException
     *             The exception is thrown, if a signal or variable is read
     *             before initialized.
     */
    public final void init() throws STRLUninitializedReadException {

        int temp = this.countDelay.getValue();

        if (temp > 1) {
            this.count = temp;
        } else {
            this.count = 1;
        }
        this.left = this.count;
        this.lastResult = STRLDelayExpression.UNKNOWN;

    }

    /**
     * Evaluates the delayExpression but ignores the count delay.
     * @param firstInstant
     *            true, if this is the firstInstant of the corresponding cell.
     * @param fresh
     *            fresh signals
     * @param unknown
     *            unknown signals
     * @return NOTSATISFIED or UNKNOWN.
     */
    public final int potentialEvaluate(final boolean firstInstant,
            final Collection fresh, final Collection unknown) {
        int result;
        if ((!this.immediateFlag) && firstInstant) {
            result = STRLDelayExpression.NOTSATISFIED;
        } else {
            int code = this.signalExp.evaluate(fresh, unknown);
            if (code == STRLBooleanExpression.FALSE) {
                result = STRLDelayExpression.NOTSATISFIED;
            } else if (code == STRLBooleanExpression.TRUE) {
                if (this.countDelay instanceof STRLIntegerConstant) {
                    try {
                        int value = this.countDelay.getValue();
                        if (value == 1) {
                            result = STRLDelayExpression.SATISFIED;
                        } else {
                            result = STRLDelayExpression.UNKNOWN;
                        }
                    } catch (STRLException e) {
                        result = STRLDelayExpression.UNKNOWN;
                    }
                } else {
                    result = STRLDelayExpression.UNKNOWN;
                }

            } else {
                result = STRLDelayExpression.UNKNOWN;
            }
        }
        return result;
    }

    /**
     * Evaluates the delay expression.
     * @param firstInstant
     *            true, if this is the firstInstant of the corresponding cell.
     * @param fresh
     *            fresh signals
     * @param unknown
     *            unknown signals
     * @return SATISFIED, NOTSATIESFIED or UNKNOWN.
     */
    public final int evaluate(final boolean firstInstant,
            final Collection fresh, final Collection unknown) {
        if (this.lastResult != STRLDelayExpression.UNKNOWN) {
            return this.lastResult;
        }
        int result;
        if ((!this.immediateFlag) && firstInstant) {
            result = STRLDelayExpression.NOTSATISFIED;
        } else {
            int code = this.signalExp.evaluate(fresh, unknown);
            if (code == STRLBooleanExpression.FALSE) {
                result = STRLDelayExpression.NOTSATISFIED;
            } else if (code == STRLBooleanExpression.TRUE) {
                this.left--;
                if (this.left == 0) {
                    result = STRLDelayExpression.SATISFIED;
                } else {
                    result = STRLDelayExpression.NOTSATISFIED;
                }
            } else {
                result = STRLDelayExpression.UNKNOWN;
            }
        }
        this.lastResult = result;
        return result;
    }

    /**
     * Returns the immediate flag.
     * @return Returns the immediate flag.
     */
    public final boolean isImmediate() {
        return this.immediateFlag;
    }

    /**
     * Returns the signal expression.
     * @return Returns the signal expression.
     */
    public final STRLBooleanExpression getExp() {
        return this.signalExp;
    }
}
```

.10.2 STRLImmediateTrans.java

```java
/*
 * Created on 13.06.2004
 */
package kiel.simulator.syncchart.transition;

import kiel.dataStructure.Transition;

/**
 * Immediate transitions have a pseudo cell as source.
 * @author Andre Ohlhoff
 */
public class STRLImmediateTrans extends STRLTransition {

    /**
     * Constructs a immediate transition with a reference to the corresponding
     * kiel transition.
     * @param t
     *            kiel transition
     */
    public STRLImmediateTrans(final Transition t) {
        super(t);
    }
}
```

.10.3 STRLNormalTermination.java

```java
/*
 * Created on 14.05.2004
 */
package kiel.simulator.syncchart.transition;

import kiel.dataStructure.Transition;

/**
 * NormalTermination transitions have only an optional effect. NormalTermination
 * transitions are enabled of the body of a cell is evaluated and returns DEAD.
 * @author Andre Ohlhoff
 */
public class STRLNormalTermination extends STRLTransition {

    /**
     * Constructs a NormalTermination with a reference to the corresponding kiel
     * transition.
     * @param t
     *           kiel transition
     */
    public STRLNormalTermination(final Transition t) {
        super(t);
    }
}
```

.10.4 STRLStrongAbortion.java

```
/*
 * Created on 14.05.2004
 */
package kiel.simulator.syncchart.transition;

import kiel.dataStructure.Transition;

/**
 * StrongAbortion transition are evaluated before evaluating the body of a cell.
 * @author Andre Ohlhoff
 */
public class STRLStrongAbortion extends STRLTransition {
   /**
    * Constructs a StrongAbortion with a reference to the corresponding kiel
    * transition.
    * @param t
    *            kiel transition
    */
   public STRLStrongAbortion(final Transition t) {
      super(t);
   }
}
```

.10.5 STRLTransition.java

```
/*
 * Created on 14.05.2004
 */
package kiel.simulator.syncchart.transition;

import java.util.Collection;
import java.util.HashSet;

import kiel.dataStructure.Transition;
import kiel.simulator.syncchart.STRLComponent;
import kiel.simulator.syncchart.STRLReactiveCell;
import kiel.simulator.syncchart.STRLSimulator;
import kiel.simulator.syncchart.action.STRLActions;
import kiel.simulator.syncchart.boolexp.STRLBooleanExpression;
import kiel.simulator.syncchart.boolexp.STRLTrue;
import kiel.simulator.syncchart.exceptions.STRLException;
import kiel.simulator.syncchart.intexp.STRLIntegerExpression;

/**
 * A transition consists of a target cell, a delay expression, a boolean
 * condition and a actions object as effect.
 * @author Andre Ohlhoff
 */
public abstract class STRLTransition extends STRLComponent {

    /**
     * this constant is used as return value for a not enabled transition.
     */
    public static final int NOTENABLED = STRLDelayExpression.NOTSATISFIED;

    /**
     * this constant is used as return value for an enabled transition.
     */
    public static final int ENABLED = STRLDelayExpression.SATISFIED;

    /**
     * this constant is used as return value, if it is not determinalbe if the
     * transition must be enabled.
     */
    public static final int UNKNOWN = STRLDelayExpression.UNKNOWN;

    /**
     * the delay expression.
     */
    private STRLDelayExpression delayExp = new STRLDelayExpression();

    /**
     * the boolean guard.
     */
    private STRLBooleanExpression guard = new STRLTrue();

    /**
     * actions as effect.
     */
    private STRLActions effect = new STRLActions();
```

```
/**
 * target of the transition.
 */
private STRLReactiveCell target = null;

/**
 * is true, if the target cell is entered through a history connector.
 */
private boolean entersHistory = false;

/**
 * is true, if the transition was initialized.
 */
private boolean isInitialized = false;

/**
 * reference to the corresponding kiel transition.
 */
private Transition kt = null;

/**
 * Constructs a transition with a reference to the corresponding kiel
 * transition.
 * @param t kiel transition
 */
public STRLTransition(final Transition t) {
    this.kt = t;
}

/**
 * Sets the kiel transition.
 * @param t kiel transition to set
 */
public final void setTransition(final Transition t) {
    this.kt = t;
}

/**
 * Returns the kiel transition.
 * @return Returns the kiel transition.
 */
public final Transition getKielTransition() {
    return this.kt;
}

/**
 * Returns true, if the target cell is entered through a history connector.
 * @return Returns true, if the target cell is entered through a history
 *         connector.
 */
public final boolean entersHistory() {
    return this.entersHistory;
}

/**
 * Sets the entershistory flag to the given value.
```

```
     * @param flag true if the target cell is entered through a history
     *            connector.
     */
    public final void setEntersHistory(final boolean flag) {
        this.entersHistory = flag;
    }

    /**
     * Sets the delay expression.
     * @param dE the delay expression to set
     */
    public final void setDelayExp(final STRLDelayExpression dE) {
        this.delayExp = dE;
    }

    /**
     * Sets the actions.
     * @param e actions
     */
    public final void setActions(final STRLActions e) {
        this.effect = e;
    }

    /**
     * Sets the guard.
     * @param g boolean expression as guard
     */
    public final void setGuard(final STRLBooleanExpression g) {
        this.guard = g;
    }

    /**
     * Sets the target cell.
     * @param t target cell
     */
    public final void setTarget(final STRLReactiveCell t) {
        this.target = t;
    }

    /**
     * Sets the context for the effect.
     * @param sim simulator as context
     */
    public final void setComponentsContext(final STRLSimulator sim) {
        this.effect.setContext(sim);
    }

    /**
     * Executes the effect.
     * @throws STRLException The exceptions is thrown if an action throws a
     *             exception during execution.
     */
    public final void execute() throws STRLException {
        this.getContext().logExecuteTransition(this.getKielTransition());
        this.effect.execute();
    }

    /**
     * Returns the target cell.
     * @return Returns the target cell.
     */
    public final STRLReactiveCell getTarget() {
        return this.target;
    }

    /**
     * Resets the delay expression and the effect.
     * @throws STRLException The exception is thrown if something went wrong.
     */
    public final void init() throws STRLException {
        this.isInitialized = true;
        this.delayExp.init();
        this.getContext().debug(
                "initialized␣transition:␣"
                        + this.getKielTransition().toString());
        this.getContext().somethingChanged();
    }

    /**
     * Returns the set of potentially emitted signals. The set contains all
     * signals, which can be emitted by the effect and also the complete
     * potential of the target cell.
     * @param fresh fresh signals
     * @param unknown unknown signals
     * @return Returns the collection of potantially emitted signals.
     */
    public final Collection getPotential(final Collection fresh,
            final Collection unknown) {
        HashSet result = new HashSet();
        result.addAll(this.effect.getPotential());
        result.addAll(this.target.getCompletePotential(fresh, unknown));
        return result;
    }

    /**
     * Evaluates whether the transition can be enabled.
     * @param firstInstant true, if the source cell is activated in this
     *            instant.
     * @param fresh fresh signals
     * @param unknown unknown signals
     * @return ENABLED, NOTENABLED or UNKNOWN.
     */
    public final int evaluate(final boolean firstInstant,
            final Collection fresh, final Collection unknown) {
        int code = this.delayExp.evaluate(firstInstant, fresh, unknown);

        if (code == STRLDelayExpression.UNKNOWN) {
            return STRLTransition.UNKNOWN;
        } else if (code == STRLDelayExpression.NOTSATISFIED) {
            return STRLTransition.NOTENABLED;
        } else {
            /* code == STRLDelayExpression.SATISFIED */
            int guardCode = this.getContext().evaluateGuard(this.guard,
                    new HashSet(), new HashSet());
            if (guardCode == STRLBooleanExpression.FALSE) {
                return STRLTransition.NOTENABLED;
            } else if (guardCode == STRLBooleanExpression.TRUE) {
                return STRLTransition.ENABLED;
            } else {
                /* guardCode == STRLBooleanExpression.UNKNOWN */
                return STRLTransition.UNKNOWN;
            }
        }
    }

    /**
     * Evaluates whether the transition can be enabled. This method is only used
     * to derive potentials.
     * @param firstInstant true, if the source cell is activated in this
     *             instant.
     * @param fresh fresh signals
     * @param unknown unknown signals
     * @return NOTENABLED or UNKNOWN.
     */
    public final int potentialEvaluate(final boolean firstInstant,
            final Collection fresh, final Collection unknown) {
        int code = this.delayExp
                .potentialEvaluate(firstInstant, fresh, unknown);
        if (code == STRLDelayExpression.NOTSATISFIED) {
            return STRLTransition.NOTENABLED;
        } else if (code == STRLDelayExpression.SATISFIED) {
            if (this.guard instanceof STRLTrue) {
                return STRLTransition.ENABLED;
            } else {
                return STRLTransition.UNKNOWN;
            }
        } else {
            return STRLTransition.UNKNOWN;
        }
    }

    /**
     * Prepares this delay expression and the actions for the next instant.
     */
    public final void prepareForNext() {
        this.delayExp.prepareForNext();
    }

    /**
     * Returns the count delay expression.
     * @return Returns the count delay expression.
     */
    public final STRLIntegerExpression getInitExpression() {
        return this.delayExp.getCountDelay();
    }

    /**
     * Returns true, if the transition was initialized.
     * @return Returns true, if the transition was initialized.
     */
    public final boolean isInitialized() {
        return this.isInitialized;
    }

    /**
     * Sets the isInitialized flag to false.
     */
    public final void setUnitialized() {
        this.isInitialized = false;
    }
}
```

.10.6 STRLTransitionHandler.java

```
/*
 * Created on 15.09.2004
 */
package kiel.simulator.syncchart.transition;

import java.util.Collection;
import java.util.HashSet;
import java.util.Iterator;
import java.util.LinkedList;

import kiel.simulator.syncchart.STRLComponent;
import kiel.simulator.syncchart.STRLReactiveCell;
import kiel.simulator.syncchart.STRLSimulator;
import kiel.simulator.syncchart.action.STRLAction;
import kiel.simulator.syncchart.action.STRLActions;
import kiel.simulator.syncchart.action.STRLInitializeTransition;
import kiel.simulator.syncchart.exceptions.STRLException;

/**
 * The TranstionHandler takes care of a list of transitions of same type for a
 * cell.
 * @author Andre Ohlhoff
 */
public class STRLTransitionHandler extends STRLComponent {

    /**
     * constant used as return value, if the handler has found a transition to
     * enable.
     */
    public static final int TRUE = 0;

    /**
     * constant used as return value, if no transition can be enabled.
     */
    public static final int FALSE = 1;

    /**
     * constant used as return value, if the handler cannot decide whether to
     * enabled a transition or not.
     */
    public static final int UNKNOWN = 2;

    /**
     * is true, if during the computation of the potential a transition must be
     * activated.
     */
    private boolean sureEnabled = false;

    /**
     * is true, if during the computation of the potential a transition can be
     * enabled.
     */
    private boolean possibleEnabled = false;

    /**
     * List of transitions, ordered by priority.
     */
    private LinkedList transitions = new LinkedList();

    /**
     * actions used to initialize the transitions.
     */
    private STRLActions initActions = null;

    /**
     * transitions not evaluated yet.
     */
    private LinkedList toEvaluate = new LinkedList();

    /**
     * Sets the transitions.
     * @param t Array of transitions, sorted by priority.
     */
    public final void setTransitions(final STRLTransition[] t) {
        STRLAction[] actions = new STRLAction[t.length];
        for (int i = 0; i < t.length; i++) {
            this.transitions.addLast(t[i]);
            actions[i] = new STRLInitializeTransition(t[i]);
        }
        this.initActions = new STRLActions(actions);
    }

    /**
     * Sets the context.
     * @param sim context
     */
    public final void setComponentsContext(final STRLSimulator sim) {
        STRLComponent.setContext(sim, this.transitions);
        this.initActions.setContext(sim);
    }

    /**
     * Prepares all transitions for the next instant.
     */
    public final void prepareForNext() {
        Iterator iter = this.transitions.iterator();
        while (iter.hasNext()) {
            STRLTransition t = (STRLTransition) iter.next();
            t.prepareForNext();
        }
        this.toEvaluate.clear();
        this.toEvaluate.addAll(this.transitions);
    }

    /**
     * Initializes all transitions.
     * @throws STRLException The exception is thrown if something went wrong.
     */
    public final void init() throws STRLException {
        this.initActions.execute();
        this.toEvaluate.clear();
        this.toEvaluate.addAll(this.transitions);
```

```java
    }

    /**
     * Evaluates whether a transition is enabled or not.
     * @param firstInstant true, if the source cell is activated in this
     *           instant.
     * @return TRUE, FALSE or UNKNOWN.
     */
    public final int evaluate(final boolean firstInstant) {
        if (!this.allInitialized()) {
            this.getContext().debug("not␣initialized");
            return STRLTransitionHandler.UNKNOWN;
        }
        while (!(this.toEvaluate.isEmpty())) {
            STRLTransition t = (STRLTransition) this.toEvaluate.getFirst();
            this.getContext().logTestTransition(t.getKielTransition());
            int code = t.evaluate(firstInstant, new HashSet(), new HashSet());
            if (code == STRLTransition.UNKNOWN) {
                this.getContext().debug("....␣is␣unknown");
                return STRLTransitionHandler.UNKNOWN;
            } else if (code == STRLTransition.ENABLED) {
                this.getContext().debug("....␣is␣enabled");
                return STRLTransitionHandler.TRUE;
            }
            this.getContext().debug("....␣is␣not␣enabled");
            this.toEvaluate.removeFirst();
        }
        return STRLTransitionHandler.FALSE;
    }

    /**
     * Returns the sureEnabled flag.
     * @return Returns the sureEnabled flag.
     */
    public final boolean isSureEnabled() {
        return this.sureEnabled;
    }

    /**
     * Returns the possibleEnabled flag.
     * @return Returns the possibleEnabled flag.
     */
    public final boolean isPossibleEnabled() {
        return this.possibleEnabled;
    }

    /**
     * Returns the target cell of the enabled transition.
     * @return Returns the target cell of the enabled transition.
     */
    public final STRLReactiveCell nextCell() {
        return ((STRLTransition) this.toEvaluate.getFirst()).getTarget();
    }

    /**
     * Executes the enabled transition.
     * @throws STRLException The exception is thrown if something went wrong.
     */
    public final void execute() throws STRLException {
        STRLTransition t = (STRLTransition) this.toEvaluate.getFirst();
        t.execute();
        this.nextCell().enter(t.entersHistory());
    }

    /**
     * Returns the potential of the transitions.
     * @param isFirstInstant true, if the source cell is activated in this
     *           instant.
     * @param fresh fresh signals
     * @param unknown unknown signals
     * @return Returns the potential of the transitions.
     */
    public final Collection getPotential(final boolean isFirstInstant,
            final Collection fresh, final Collection unknown) {

        this.sureEnabled = false;
        this.possibleEnabled = false;

        HashSet result = new HashSet();

        Iterator iter = this.transitions.iterator();

        while (iter.hasNext()) {
            STRLTransition t = (STRLTransition) iter.next();
            int code = t.potentialEvaluate(isFirstInstant, fresh, unknown);
            if (code == STRLTransition.NOTENABLED) {
                continue;
            }
            this.possibleEnabled = true;
            result.addAll(t.getPotential(fresh, unknown));
            if (code == STRLTransition.ENABLED) {
                this.sureEnabled = true;
                break;
            }
        }
        return result;
    }

    /**
     * Returns true, if all transitions have been initialized. Is used to delay
     * evaluation until all transitions are initialized.
     * @return Returns true, if all transitions have been initialized.
     */
    private boolean allInitialized() {
        Iterator iter = this.transitions.iterator();
        while (iter.hasNext()) {
            if (!((STRLTransition) iter.next()).isInitialized()) {
                return false;
            }
        }
        return true;
    }
}
```

.10.7 STRLWeakAbortion.java

```
/*
 * Created on 14.05.2004
 */
package kiel.simulator.syncchart.transition;

import kiel.dataStructure.Transition;

/**
 * WeakAbortion transitions are evaluated after the evaluation of the body of a
 * cell.
 * @author Andre Ohlhoff
 */
public class STRLWeakAbortion extends STRLTransition {

    /**
     * Constructs a WeakAbortion with a reference to the corresponding kiel
     * transition.
     * @param t
     *            kiel transition
     */
    public STRLWeakAbortion(final Transition t) {
        super(t);
    }
}
```