

Universität des Saarlandes
Naturwissenschaftlich-Technische Fakultät I
Fachrichtung 6.2 - Informatik
Dependable Systems and Software

Diplomarbeit

An Eclipse plugin for MoDeST

Christophe Bouter

May 2007

Angefertigt unter der Leitung von:
Prof. Dr.-Ing. Holger Hermanns

Erklärung

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

Saarbrücken, im Mai 2007

Christophe Bouter

I would like to thank Prof. Holger Hermanns for giving me the thesis which proved to be much more challenging than I expected and for his great support during the whole process of completion. Further I would like to thank Niko Paltzer and Patrick Pekczynski for sharing ideas with me. I would like to thank the people from the ANTLR-interest mailinglist who helped me out of some real problems with the parser. A special thanks goes to Reza Pulungan who always had an ear for my MoDeST problems.

I'm deeply indebted to my family who supported me greatly during my studies and to my girlfriend who was very patient the last months during the writing of this thesis.

Abstract

Many programming tools and specific editors have seen the light the past years. One widely known and standard for the Java language is Eclipse. Eclipse is open source, built with Java and composed of plugins. The choice for a plugin based architecture was taken in order to ease the maintenance load and to give a big community the possibility to build their own plugins for Eclipse. Since the support for the MoDeST language was not at the best for the programmers, a new plugin for Eclipse which supports the programmer by his development of MoDeST models was realized. In this thesis a short introduction to MoDeST and the Eclipse plugin will be given before having a deeper look into the implementation of the plugin.

Contents

| | | |
|----------|---|-----------|
| 1 | Introduction | 1 |
| 2 | A step-by-step guide through the plugin features | 3 |
| 2.1 | The editor | 3 |
| 2.2 | Plugin preferences | 4 |
| 2.3 | External programs | 6 |
| 2.4 | Simulating MoDeST in Eclipse | 7 |
| 2.4.1 | Short introduction to MoDeST | 7 |
| 2.4.2 | The step simulation | 10 |
| 2.5 | Installation | 18 |
| 3 | Parsing MoDeST | 19 |
| 3.1 | Lexer and Parser | 19 |
| 3.2 | The outline Walker | 20 |
| 3.3 | The simulation Walker | 20 |
| 4 | The editor | 22 |
| 4.1 | The main classes | 22 |
| 4.1.1 | ModestEditorPlugin | 22 |
| 4.1.2 | ModestEditor | 22 |
| 4.2 | Syntax highlighting and content assistance | 24 |
| 4.2.1 | Syntax highlighting | 24 |
| 4.2.2 | Content assistance | 24 |
| 4.3 | Error markup | 25 |
| 4.4 | Outline | 25 |
| 4.4.1 | ModestContentOutlinePage | 25 |
| 4.4.2 | Models | 26 |
| 4.5 | Editor Preferences | 27 |
| 4.5.1 | Default initialization | 27 |
| 4.5.2 | ModestPreferencePage | 28 |

| | | |
|----------|--|-----------|
| 5 | Launch framework integration | 29 |
| 5.1 | The main classes | 29 |
| 5.1.1 | ModestLaunchPlugin | 30 |
| 5.1.2 | ModestLaunchDelegate | 30 |
| 5.2 | Launch Preferences | 30 |
| 5.2.1 | ModestLaunchPreferencePage | 30 |
| 5.2.2 | ModestLaunchTab | 30 |
| 6 | Step simulation | 32 |
| 6.1 | The simulation theory | 32 |
| 6.1.1 | The MoDeST core | 33 |
| 6.1.2 | Processes and exception handling | 34 |
| 6.1.3 | Relabeling | 34 |
| 6.1.4 | palt, guards and assignments | 35 |
| 6.2 | The simulation implementation | 36 |
| 6.2.1 | The framework | 36 |
| 6.2.2 | The MoDeST core | 38 |
| 6.2.3 | Processes and exception handling | 39 |
| 6.2.4 | Relabeling | 41 |
| 6.2.5 | palt, guards and assignments | 42 |
| 6.2.6 | Backtracking | 42 |
| 7 | Extension points used | 45 |
| 7.1 | modesteditor.core | 45 |
| 7.1.1 | org.eclipse.ui.editors | 45 |
| 7.1.2 | org.eclipse.core.filebuffers.documentSetup | 46 |
| 7.1.3 | org.eclipse.ui.preferencePages | 46 |
| 7.1.4 | org.eclipse.ui.views | 47 |
| 7.2 | modesteditor.launch | 47 |
| 7.2.1 | org.eclipse.ui.preferencePages | 47 |
| 7.2.2 | org.eclipse.debug.core.launchConfigurationTypes | 48 |
| 7.2.3 | org.eclipse.debug.ui.launchConfigurationTabGroups | 48 |
| 7.2.4 | org.eclipse.debug.ui.launchConfigurationTypeImages | 49 |
| 8 | Conclusion | 50 |

1 Introduction

This thesis presents the work done towards an integration of MoDeST in the Eclipse framework.

MoDeST

MoDeST is a modelling and description language for stochastic timed systems, which was developed at the university of Twente [4]. It is used to model probabilistic non-deterministic systems with realtime constraints. The thus obtained models can be used with a model-checker in order to be verified or can be analyzed with a discrete event simulation. The semantic model which serves as a base for the language are so called STAs (*Stochastic Timed Automata*).

The MoToR tool [16] (*MOdest TOol enviRonment*) gives us the means to simulate MoDeST specifications and to analyze the obtained results. The MoDeST syntax is kept close to the C syntax in order to simplify the first steps. In addition we find a `do` construct for loops, a construct for branching (`alt`), probabilistic branching (`pa1t`), blocking guards (`when`) and synchronized concurrency (`par`).

Eclipse

Eclipse [6] is an open source community whose projects are focused on building an open development platform comprised of extensible frameworks, tools and runtimes for building, deploying and managing software across the lifecycle. A large and vibrant ecosystem of major technology vendors, innovative start-ups, universities, research institutions and individuals extend, complement and support the Eclipse platform.

Eclipse is mainly known for being a full-featured Java IDE (*Integrated Development Environment*). Due to the philosophy behind Eclipse, basically every feature is encapsulated into a separate plugin, even the Java IDE is a collection of plugins and is not part of the Eclipse core.

Problem statement

When implementing a big or complex program, the programmer will find help with tools which simplify his work by giving some features like completion and syntax highlighting. The better ones even give support when debugging a program that doesn't run the way it was intended.

This kind of support isn't given for MoDeST [4], one could only use a text editor or the editor supplemented by MoToR [16] without even syntax highlighting. Debug support wasn't provided either or only in a rudimentary way by the compiler that wasn't much help.

When one thinks about tools which provide all the above mentioned means of help for a programmer, one has to think about Eclipse [6] which gives all this and much more in the context of the Java programming language [13]. So it was an obvious step to try to get support for MoDeST into Eclipse in order to provide all or at least many of the features a programmer wants, in order to give the support to a MoDeST-programmer which it lacks as of today.

Building an Eclipse plugin

Eclipse plugins are built using Eclipse itself. Eclipse helps the developer by providing a wizard which sets up the basics every plugin needs. The integration of the newly built plugin is done through an XML file named *plugin.xml*. This file contains all the informations needed to run the plugin in Eclipse. The main informations are the plugin ID, the plugin version and the activator class. In addition to those informations all extension points (c.f. 7) appearing in the plugin are also recorded in the *plugin.xml* file. The further implementation of the plugin is done through Java classes and packages as known from any other implementation.

After the implementation of the plugins, they are packaged into one or more features and those features are distributed through Eclipse update sites [10]. The update site of the MoDeST plugin and a short manual can be found on the plugin homepage [14].

Overview

Chapter 2 guides the user step by step through all the features of the MoDeST plugin for Eclipse. Since not all features are only based on Eclipse but for example the step simulation needs a certain understanding of the MoDeST language, MoDeST is introduced on hand of an example. This example contains all the central language constructs and should be a good starting point for MoDeST beginners. Chapter 3 gives an overview over the ANTLR [1] based parsing framework that is used in the editor and the step simulation. This chapter explains the choice of the parser generator and gives some details about the implementation of the lexer, parser and the walker used to transform the abstract syntax tree. In chapter 4 is pictured how the features seen in chapter 2 are built and how the MoDeST specific editor and its sideparts fits into the Eclipse framework. Chapter 5 pictures shortly how the already existing toolchain for MoDeST was integrated in the plugin via the Eclipse launch framework [17]. An example run of the step simulation of the MoDeST plugin was shown in chapter 2. In Chapter 6 the theory standing behind this simulation is presented before explaining how the theory model was implemented and integrated into the Eclipse framework. The theory model as well as the implementation are developed in an incremental fashion to ease the understanding. In chapter 7 a reference of the extension points used in the build process of the plugin is given. This should facilitate the entry of other Eclipse developers into the implementation of the plugin. Chapter 8 gives a summary over the work presented in this thesis and shows a short outlook of further work that could be done in relation with the MoDeST plugin for Eclipse.

2 A step-by-step guide through the plugin features

This chapter shall present step-by-step all features present in the MoDeST plugin for Eclipse. First the editor and the related parts are shown before going over to the plugin specific preference pages. After that an introduction on how to use the MoDeST specific external programs is given. This is followed by an initiation to the MoDeST language through an example presenting the primary language features and explaining their semantics. The guide is closed by an example how MoDeST code can be simulated in the context of the MoDeST plugin. Last, some installation instructions are given.

2.1 The editor

Now that the plugin is installed the focus will be placed onto the features and the restrictions of the plugin and the Eclipse framework. To achieve this an example will build a MoDeST program from the scratch. In order to use some Eclipse specific mechanisms (for example Markers, and lots of Markers are used) the file where the program will be written in needs to be placed in the Workspace. So first a project should be created by using the “File” menu and choosing “New” and “Project”. For a MoDeST project a “General Project” is suited. After having named the project it appears in the package explorer. By right clicking this project it is now possible to create a “New” “File” with a ‘.modest’ ending. This is necessary to trigger the activation of the MoDeST specific editor of the MoDeST plugin.

Once the .modest file is created the editing can start. The editor disposes of the usual features one would expect such as syntax highlighting and error markup (Note: the credited syntax errors are the same the MoToR tool would find). To alleviate the user a text completion is present. This completion spans all MoDeST keywords and the user introduced variables. After having started a word the user can activate the completion by pressing *CTRL+SPACE* (c.f. 2.2).

The afore mentioned error markup does not only mark syntax errors (red Marker with red squiggles) but also shows unused variables to the user (yellow Marker with yellow squiggles).

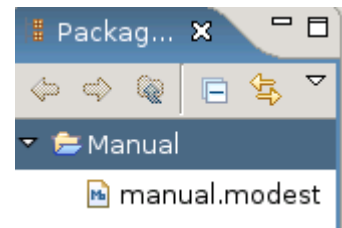


Figure 2.1: Explorer with MoDeST file

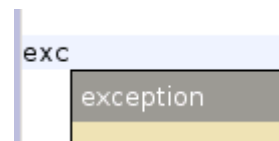


Figure 2.2: Activated completion

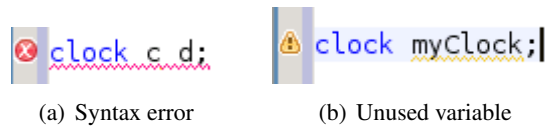


Figure 2.3: Error Marker

2.2 Plugin preferences

In order to give the user the possibility to customize some aspects of the MoDeST plugin two preference pages were introduced. In the first, general, one [2.4](#) the user can change the colors of the syntax highlighting via the color fields for every category of keywords and disable the syntax error markup via a checkbox. By default the markup is activated and the colors are chosen a way they incorporate quite good in the general look and feel of the Eclipse framework.

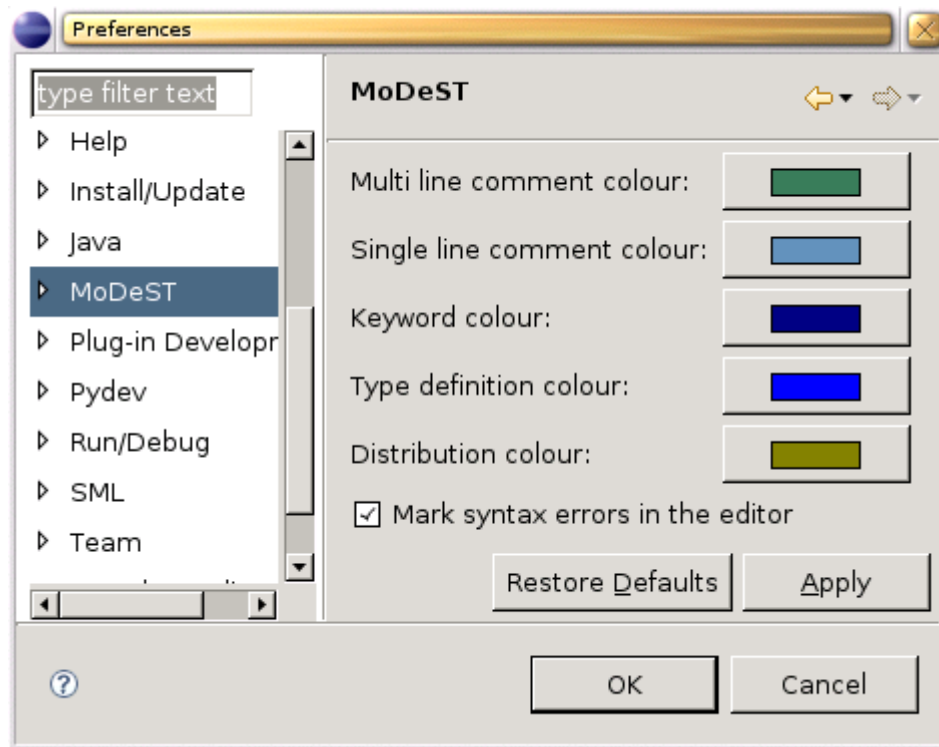


Figure 2.4: Main preferences for the MoDeST plugin

In the second preference page [2.5](#) the preferences for the launching of external programs (c.f. [2.3](#)) are put together. There are for instance the locations of the MoDeST compiler (the `momodest` binary) and the FSNS interface and the path to the destination folder of the output of all external

programs. Further there is a checkbox that says “Generate dot file”, if it is checked by default the compiler will not compile the file but generate a dot file holding the representation of the LTS (*Labelled Transition System*) of the program.

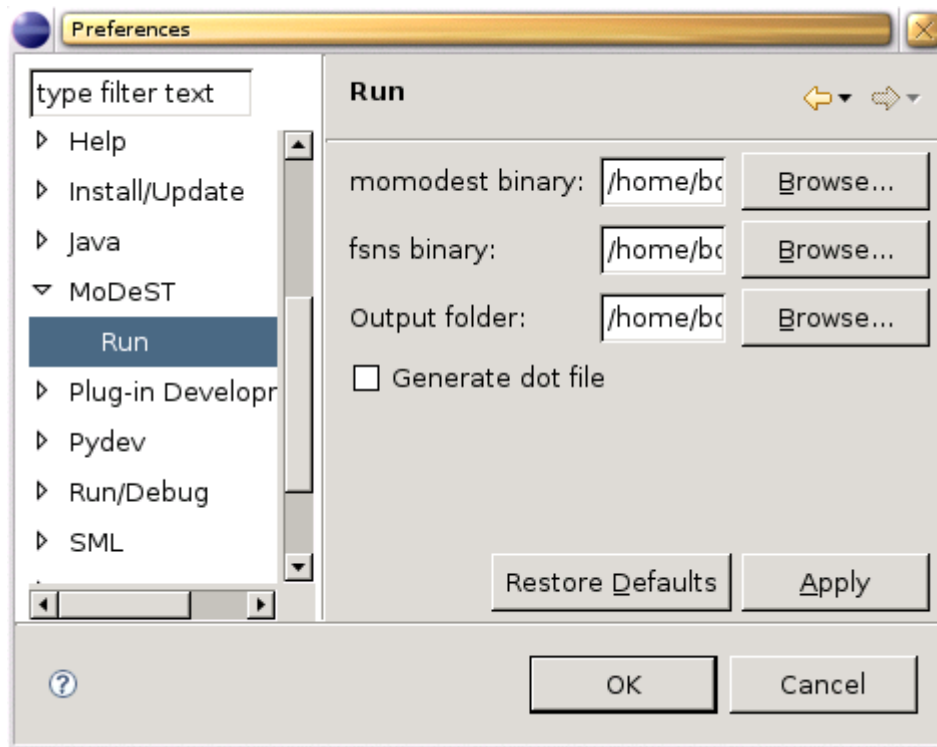


Figure 2.5: External program preferences

2.3 External programs

To start the MoDeST related external programs the “Run” dialog must be started as shown beside. In this dialog a new configuration must be created in the “Modest” group by right clicking the group and choosing “New”.

In this new launch tab the project and the file have to be entered manually. The user has then the possibility to choose whether to run the compiler or get a dot output or run the FSNS (*First State Next State*) interface. The dot file checkbox is set according to the setting in the preference page (c.f. 2.5).

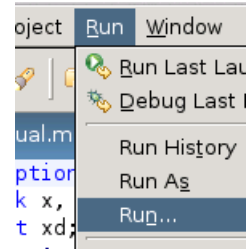


Figure 2.6: The run menu

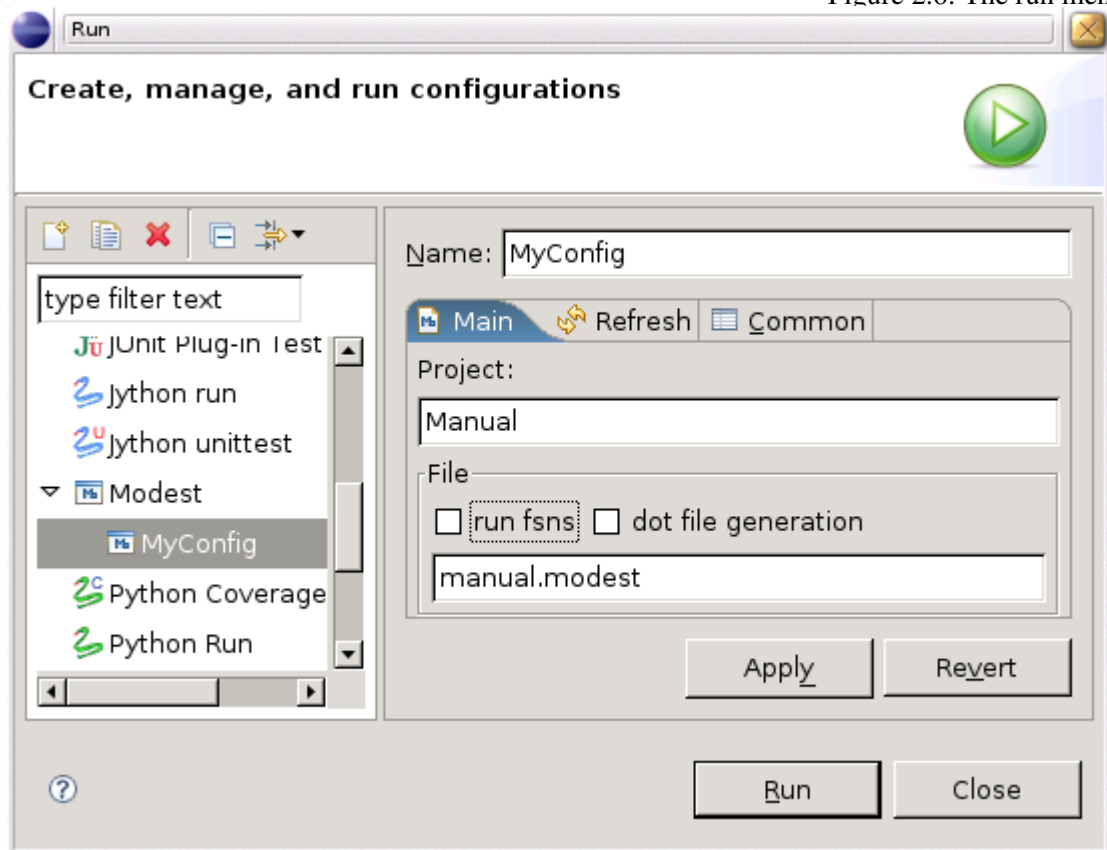
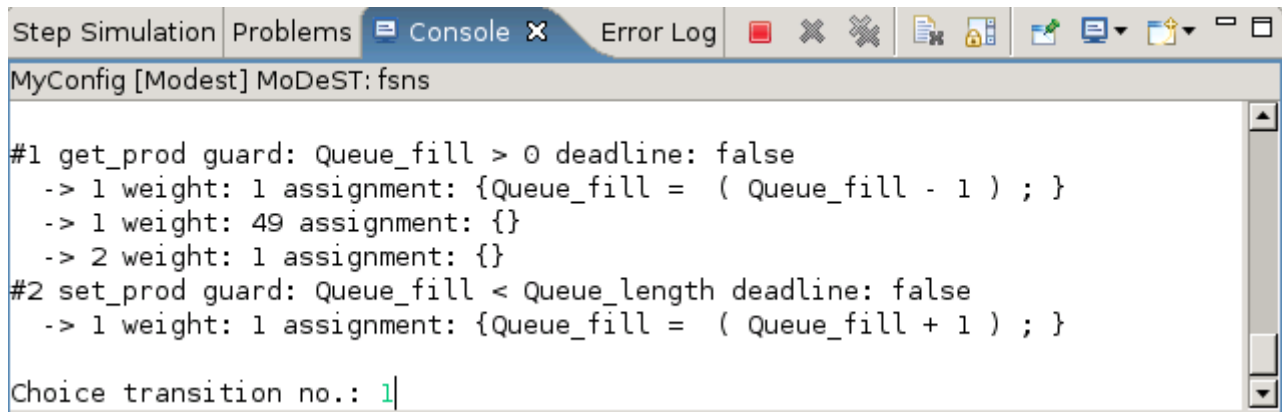


Figure 2.7: Launch tab

When the FSNS checkbox is set and the interface starts after clicking the “Run” button the Eclipse console View is used to display the output of the interface. The console is also used to gather the input of the user and is thus interactive. The output on the standard output stream is

black, on standard error red and the input is displayed in green as seen in 2.8.



```

Step Simulation Problems Console x Error Log
MyConfig [Modest] MoDeST: fsns

#1 get_prod guard: Queue_fill > 0 deadline: false
  -> 1 weight: 1 assignment: {Queue_fill = ( Queue_fill - 1 ) ; }
  -> 1 weight: 49 assignment: {}
  -> 2 weight: 1 assignment: {}
#2 set_prod guard: Queue_fill < Queue_length deadline: false
  -> 1 weight: 1 assignment: {Queue_fill = ( Queue_fill + 1 ) ; }

Choice transition no.: 1|

```

Figure 2.8: The FSNS interface in the console View

2.4 Simulating MoDeST in Eclipse

As seen in figure 2.8 some means to simulate MoDeST code in a stepwise fashion already exists. But the figure 2.8 shows also the drawbacks of this simulation, the user only views the transitions and cannot see where exactly the simulation is in the entered code. Therefore a step simulation was added to the MoDeST plugin that can highlight the active code of the transition. So that even beginner can follow the example used to demonstrate the functionality of the simulation, an introduction of the MoDeST language is given first. The reader may not want to be overwhelmed by theory therefore the introduction is done via an example that comprises all the main language constructs. The example handled is by no means exhaustive in the presentation of the language, for a complete introduction refer to [5]. After the reader has gained a basic understanding of the MoDeST language, this example is simulated presenting the main language features. The course of the simulation will be shown on hand of pictures that explain clearly what happens in the simulation.

2.4.1 Short introduction to MoDeST

This introduction will be based on an example of a cashier in a discount market environment. The code representing the cashier can be seen in figure 2.9. This example is a slight adaptation of an example presented in [3].

```

exception no_price;
clock x, y; (5)
float xd; (4)
patient action get_prod, cash, set_price;

patient action set_prod;

process Cashing() { (1)
(6){= xd=Uniform(10,20), x=0 =};(3)
  urgent (x >= xd) (9)
  when (x>= xd) (8)
  cash (7)
}

par { (2)
  :: Arrivals()
  :: Queue(3)
  :: Cashier()
}

process Cashier() {
  do { :: (16)
    try { (10)
      get_prod palt { (14)
        :49: Cashing()
        (15):1: throw no_price
        (11)
      }
    } catch (no_price) { (12)
      y=0;
      urgent(y>=240)
      when(y>=120)
      set_price;
      Cashing() (13)
    }
  }
}

```

Figure 2.9: The cashier code

Syntax introduction

The MoDeST language allows to specify *processes* (1). They can be either composed in *parallel* with a *par* operator (2) in order to model concurrency or composed *sequentially* with a *;* operator. Processes can manipulate *data variables* by *assignments* (3). Data variables are typed and must be declared before their use, the point of declaration (4) determines their *scope*. These variables can be *local* to a process or *global*. A particular type of variable which can be declared is the *clock* type (5). Clocks may be read like an ordinary float variable and reset to zero by assignment, but they advance their value linear to system time. All clocks run at the same speed. MoDeST provides the means to sample values from a predefined set of probability distributions. At (6) one observes that *xd* is assigned a sample from the uniform distribution over the interval [10,20].

Processes aren't restricted to manipulate data but they can also interact with other parallel processes (or the environment) via visible actions (7). The occurrence of these actions within a process can be guarded by a *when(.)* clause (8), specifying a boolean enabledness condition. These clauses may also refer to clock values in the condition. In addition, an *urgent(.)* clause (9) allows one to put a deadline on the latest occurrence of an action after which the action has to be taken.

Processes in the body of a *par* construct (2) perform actions and assignments independently

from each other, safe for common non-local actions that need to be executed synchronously. For this synchronization a so called common alphabet is built before the execution of the parallel processes. Actions can be declared (4) either *patient* or *impatient*. When declared patient, an action has no urgency constraint and the process waits for the parallel processes to do the same action when possible. By default the actions are *patient* and thus the synchronization is blocking because no urgency is given.

MoDeST also provides means to raise and handle *exceptions*, which must be declared (4) first-hand. Within a `try` block (10) an exception may be *raised* (11), and can be *caught* (12) by a corresponding `catch` statement. The process control is then handed over to the exception handler. Another way of handing over process control is by a simple process call (13). Upon termination of the called process, the calling process gains back control, like in an ordinary procedure call. In this setting a try-catch-block has to be seen as one process.

Several *nondeterministic alternatives* can be declared via an `alt` construct (not present in the example). A variation of it, is the `paIt` construct (14), which provides a weighted *probabilistic choice*, where each *weight* has the form `:w`: (15), with w a positive natural number. A `paIt` must always be preceded by an action, either explicitly or implicitly by the *tau* action. *Loops* (16) are also present in the syntax with a `do` keyword. The body is repeated until a `break` action is encountered (not present in the example).

Semantics introduction

For a complete overview over the MoDeST semantics please refer to [5]. Here is only given a very brief introduction summarizing the most important concepts that are not common to other languages.

Most of the programmatic features in MoDeST are handled as in other programming languages. For example exception handling is very like the handling in Java [13]. First have a look at concurrency in a `par` construct. Before execution the common alphabet is built, e.g. the actions are collected that occur in more than one process. During execution the different processes are executed independently until an action is reached that is present in the common alphabet. The process of this action is then blocked until **all** other processes that have the blocked action in their alphabet are ready and can take this action. The action is then taken simultaneously and the processes resume the independent execution. The parallel process is terminated when all child processes are terminated successfully.

Another non-common feature in MoDeST is the possibility to have transitions that end in probabilistic alternatives. A picture of such a transition can be seen in figure 6.2. Such transitions are obtained with the `paIt` construct which has weighted probabilistic alternatives. Such a transition is taken by first handling the action guarding the `paIt` construct and then making a probabilistic choice over the alternatives of the `paIt` and handling the assignment in the alternative. The whole transition is done in an atomic way and cannot be split.

In the syntax introduction 2.4.1 guards were presented. These guards are written in `when(.)` clauses and should not be confused with `if`-statements known from other programming lan-

guages. The guards in MoDeST are blocking and are not dismissed if their expression evaluates to false. The program control waits at the guard until the expression evaluates to true due to an advanced clock or an assignment performed in a parallel process.

2.4.2 The step simulation

Now that the user knows the basics of the MoDeST syntax and semantics, a simulation of the code of the example 2.9 can be shown in order to demonstrate how the step simulation of the MoDeST plugin works. The code presented in the figure 2.9 is completed in the simulation by a `Queue` process modelling the products waiting to be cashed and an `Arrivals` process modelling the arrival of the products. The focus in the example simulation is placed on the cashier so that the reader may understand all that happens. The steps of the simulation are presented in seven images that show the progress of the simulation how the user would observe it in his Eclipse instance, starting with figure 2.10.

```

exception no_price;
clock x, y;
float xd;
patient action get_prod, cash, set_price;

patient action set_prod;

process Cashing() {
  [= xd=Uniform(10,20), x=0 =];
  urgent (x >= xd)
  when (x>= xd)
  cash
}

process Cashier() {
  do { ::
    try {
      get_prod palt {
        :49: Cashing()
        :1: throw no_price
      }
    } catch (no_price) {
      y=0;
      urgent(y>=240)
      when(y>=120)
      set_price;
      Cashing()
    }
  }
}

```

MoDeST Step Simulation Problems Console Error Log

Take

[fill < length] set_prod (probabilistic transition following)

[fill > 0] get_prod (probabilistic transition following)

Figure 2.10: After having started the simulation with the “Start simulation” button in the action bar, two transitions appear in the “Step Simulation View”. One with the *get_prod* action and another with the *set_prod* action.

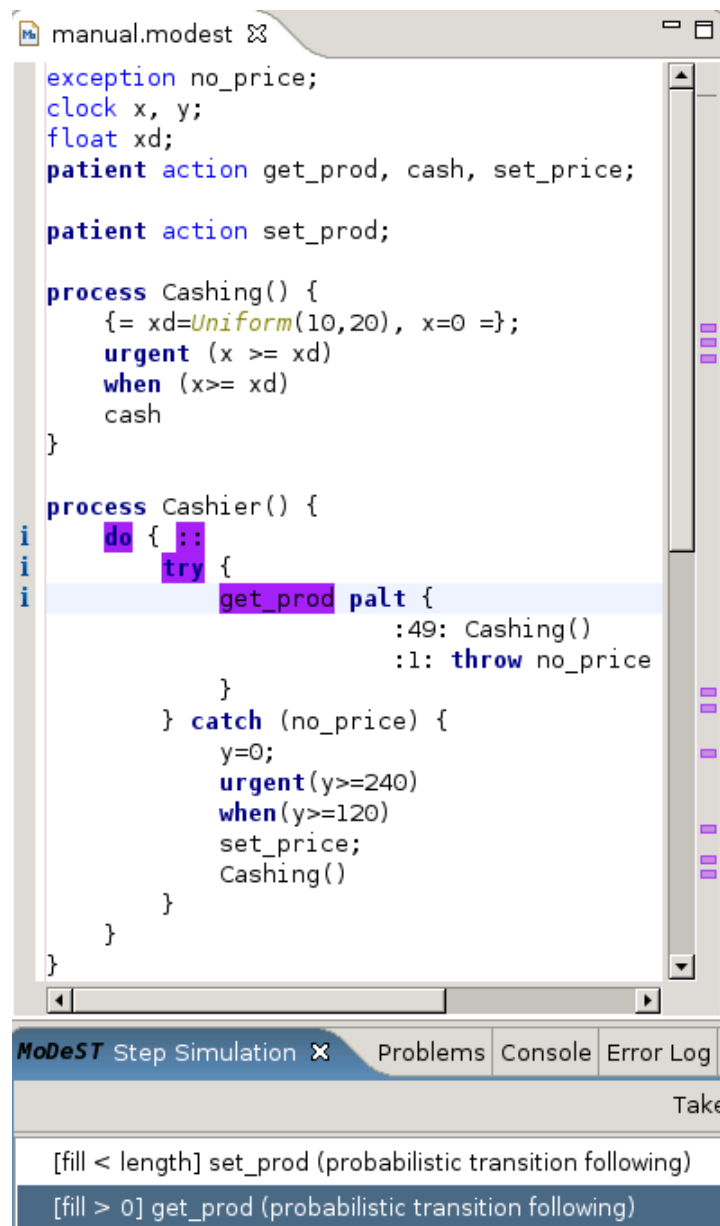


Figure 2.11: Since the interest is placed on the *Cashier* process the transition featuring the *get_prod* action is selected. Marker are set in the text that highlight the active parts of the code. **Important:** for the Markers to be highlighted as presented, the user needs to set the “Text as” attribute of the “Info Marker” in the “Preferences” under “General->Editors->Text Editors->Annotations” to “Highlight”.

The screenshot shows the Eclipse IDE with a MoDeST code editor and a simulation console. The code defines an exception, variables, and two processes: Cashing() and Cashier(). The Cashier() process contains a try-catch block for the no_price exception. The simulation console shows two probabilistic choices for the 'fill -= 1' assignment, with the second choice (0.02 probability) being selected to demonstrate exception handling.

```

manual.modest ☒
exception no_price;
clock x, y;
float xd;
patient action get_prod, cash, set_price;

patient action set_prod;

process Cashing() {
  {= xd=Uniform(10,20), x=0 =};
  urgent (x >= xd)
  when (x>= xd)
  cash
}

process Cashier() {
  do { ::
  try {
  get_prod palt {
    :49: Cashing()
    :1: throw no_price
  }
  } catch (no_price) {
    y=0;
    urgent(y>=240)
    when(y>=120)
    set_price;
    Cashing()
  }
}
}

```

MoDeST Step Simulation ☒ Problems Console Error Log

Take

Probability of the alternative 0.98
Assignment(s): fill -= 1

Probability of the alternative 0.02
Assignment(s): fill -= 1

Figure 2.12: Once the transition is taken the probabilistic choice is offered. Along with this choice the assignments are shown to the user. The second alternative with the lower probability is selected in order to present the exception handling.

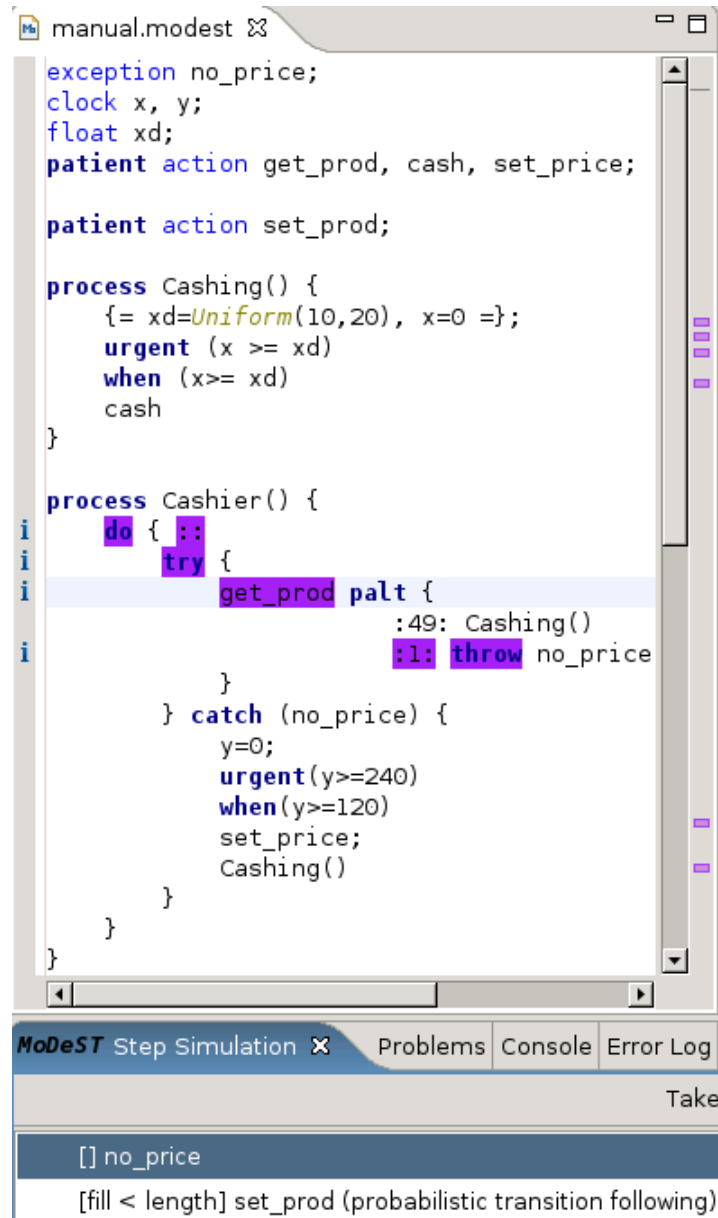


Figure 2.13: In order to show the exception handling the *throw* statement is selected since it was enabled with the probabilistic choice taken before.

```

exception no_price;
clock x, y;
float xd;
patient action get_prod, cash, set_price;

patient action set_prod;

process Cashing() {
  {= xd=Uniform(10,20), x=0 =};
  urgent (x >= xd)
  when (x >= xd)
  cash
}

process Cashier() {
  do { ::
  try {
    get_prod palt {
      :49: Cashing()
      :1: throw no_price
    }
  } catch (no_price) {
    y=0;
    urgent(y >= 240)
    when(y >= 120)
    set_price;
    Cashing()
  }
}
}

```

MoDeST Step Simulation x Problems Console Error Log

Take

[y >= 120] set_price

[fill < length] set_prod (probabilistic transition following)

Figure 2.14: The thrown exception is caught and next the first action (*set_price*) in the *catch* statement can be taken.

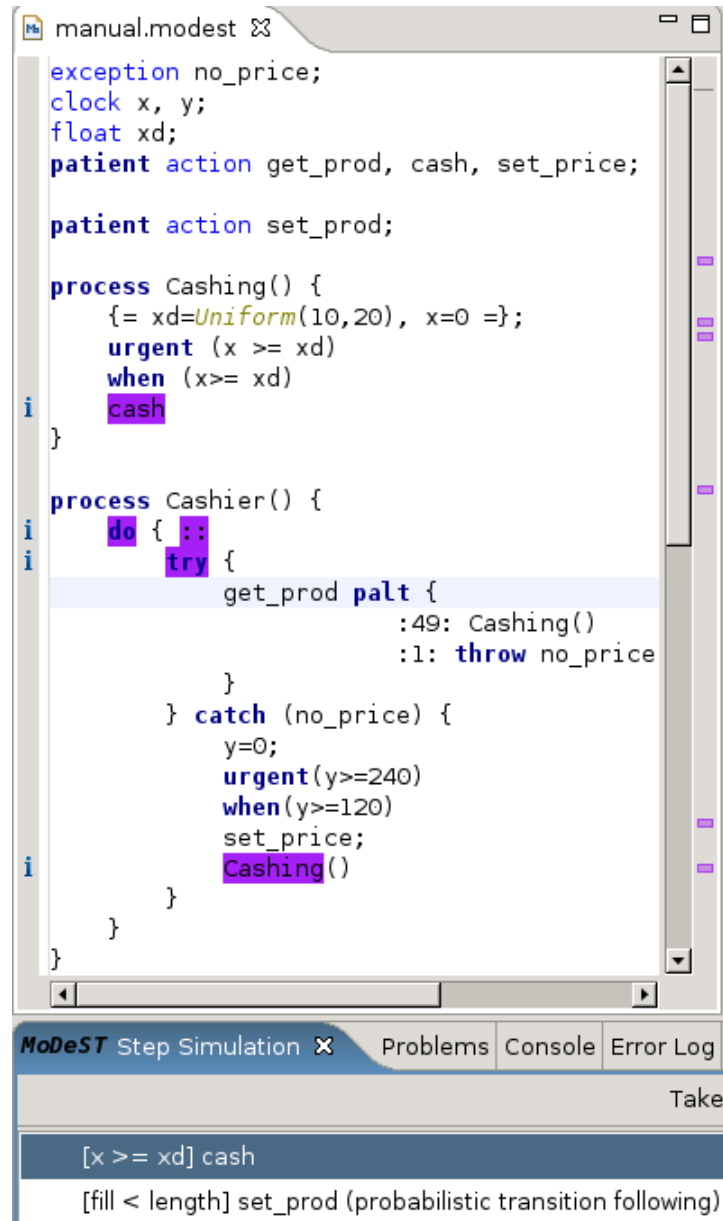


Figure 2.15: Now the process call of the *Cashing* process presents the *cash* action in a transition.


```

exception no_price;
clock x, y;
float xd;
patient action get_prod, cash, set_price;

patient action set_prod;

process Cashing() {
  [= xd=Uniform(10,20), x=0 =];
  urgent (x >= xd)
  when (x>= xd)
  cash
}

process Cashier() {
  do { ::
    try {
      get_prod palt {
        :49: Cashing()
        :1: throw no_price
      }
    } catch (no_price) {
      y=0;
      urgent(y>=240)
      when(y>=120)
      set_price;
      Cashing()
    }
  }
}

```

MoDeST Step Simulation Problems Console Error Log

Take

[fill < length] set_prod (probabilistic transition following)
 [fill > 0] get_prod (probabilistic transition following)

Figure 2.16: After the transition with the *cash* action the original state is restored as one can see on the available transitions due to the fact that the *Cashier* process is a big *do* loop.

2.5 Installation

The installation is very straightforward and uses the Eclipse update mechanism. This update mechanism is found in the “Help” menu under the point “Software Updates”. Since we want to install a new plugin we choose the menu point “Find and install”. In the newly opened window we will “Search for new features to install”. Before installing the MoDeST plugin the ANTLR plugin is required as a dependency. Therefore install this plugin first by following the instructions on the homepage [2].

Now having the required dependency the installation of the actual MoDeST plugin is possible. In the beforehand opened window “Update sites to visit” the user creates a “New remote site” with the title ‘MoDeST plugin’ and as URL the URL of the plugin [14]. This new site (‘MoDeST plugin’) should be automatically selected if not select it manually. On the “Next” screen select the topmost MoDeST feature. The “Next” screen shows a license agreement that has to be done and after that the “Finish” button is clicked and Eclipse restarted. Now the MoDeST plugin is ready to use.

3 Parsing MoDeST

Some of the features presented in chapter 2 are strongly related to the MoDeST code entered in the editor window of Eclipse. In order to make further use of this code it needs to be parsed and transformed into something machine understandable that can easily be worked with. For this purpose a parser is needed. There are many parser generators out there and probably the first choice for a Java parser generator is ANTLR [1]. Since MoToR [16] also uses an ANTLR based parser and the grammar could thus be leaned onto the MoToR MoDeST grammar, the choice was easily made. In this way a good compatibility between the MoToR tool and the MoDeST plugin can be ensured. First the lexer and parser will be introduced before explaining the two tree walker that are used in the plugin to translate the AST (*Abstract Syntax Tree*) into other tree structures put to further use.

3.1 Lexer and Parser

The lexer was kept very basic, it recognizes numbers (integers and floats), operators, comments and identifier. In the identifier section, ANTLR is told to test for the literals that occur in the parser. In this way only the identifier which doesn't match a parser literal are really transformed into identifier parser literals. Whitespaces are omitted since they are of no further need, same goes for the comments since they have no relevance for the semantics of the code.

The parser is based onto the MoToR parser which is also written with ANTLR. The rules of both parsers are kept very close to the language specification found in the documentation section of the MoToR homepage [16]. This way the greatest possible compatibility between the MoDeST plugin and the MoToR tool could be ensured. The generation rules had to be rewritten since the MoToR parser grammar has a C++ output and the plugin needs a Java output.

Already at parsing time some error checking of the code is done. First basic type checking is done by a lookup if the type of a variable declaration is in the list of built-in types or in the list of self-declared types. In order to keep these lists up to date every new type is registered in the list of self-declared types when the parser encounters a type definition. Not only types are checked for their declaration, this mechanism is applied on variables too. When a variable is used in the program code entered the parser makes a lookup to ensure that the variable was declared beforehand. Here goes the same as before, variables are registered on their declaration in order to provide the lookup.

The builtin error recognition mechanism is put to use during the detection of syntactic errors, for example a missing bracket. This error is recorded via the `reportError` method of the parser

and transformed into an `IRegion` containing the information where in the document the error is located. This list of errors is put to further use in the editor (c.f. chapter 4).

The main purpose of the parsing is to obtain an AST (*Abstract Syntax Tree*) which is a low-level tree representation of the MoDeST program parsed. This AST can then be transformed into something more useful with the help of `ASTWalker` classes. The AST representation of the code is always the same so we need only one parser. But the AST can be transformed into multiple models of the code each used for another purpose. For every such model an own `ASTWalker` is needed. The Walkers used in this thesis will be presented next. In both further uses of the AST the line and column informations are needed to get the positioning of the nodes in the tree. Since default AST node doesn't contain those informations a subclass with the positioning was needed. It was declared in the `modesteditor.core.antlrparser` package in the class `ASTWihOffsetInfos`.

3.2 The outline Walker

The first `ASTWalker` presented is the `ModestOutlineWalker`. The Walker takes the AST and transforms it into a model representation of the declarations. This model is used to display the outline of the source code in the editor. Since only the declarations are taken into account the Walker discards everything else except for the declarations.

These declarations are transformed into `Model` nodes of a tree. All the `Model` subclasses are in the package `modesteditor.core.model`. There are subclasses for the declarations of actions, exceptions, other variables, types and processes. The `Process` class is the only one which can hold child nodes since only in a process declaration other local declarations can be made. How this tree is used to build the outline will be shown in 4.4.

3.3 The simulation Walker

The second `ASTWalker` used is the `ModestSimulationTreeWalker`. The Walker builds from the AST obtained from the parser a new tree of `SimulationNode` that represents the parse tree of the entered code that contains all necessary informations such as scoping, variable names and so on. The tree structure is not built explicitly as in the outline Walker by adding to the parent the child node, but implicitly so that a child node registers itself by the parent during the initialization. How the thus obtained tree is used in the step simulation of the MoDeST code will be shown in chapter 6.

There exists subclasses of `SimulationNode` in the package `modesteditor.core.stepsimulation` for every construct in the MoDeST language. Exceptions are process call since a process call in MoDeST is nothing else than a textual replacement of the call by the process body instantiated with the argument variables. So in the Walker the tree built by the body of the process is put in place of the process call. The Walker doesn't differentiate between a weighted

alternative used in a *palt* and a “normal” alternative used in a *alt* since it’s semantically the same as stated in [5].

4 The editor

In the previous chapter the MoDeST parsing and translation tools were seen. In this chapter the custom editor for MoDeST will be presented and the first uses for the parsing tools will be demonstrated. The MoDeST specific editor is started whenever the MoDeST plugin is installed in an Eclipse instance and a file with a `.modest` ending is opened. That happens because a new editor for MoDeST files is registered on the editor extension point (c.f. 7.1.1).

In order to discuss the editor of the MoDeST plugin the startup class of the plugin will be shown before going to the actual class managing the editor. Once this is done, a discussion of the classes controlling the editor features such as syntax highlighting and error markup will follow. The chapter will be closed by an introduction to the preferences of the editor and their realisation. All classes presented in this chapter are part of the `modesteditor.core` plugin and can be found in the `modesteditor.core` package or one of its subpackages.

4.1 The main classes

4.1.1 ModestEditorPlugin

The `ModestEditorPlugin` class contained in the `modesteditor.core` package is the launcher for the MoDeST editor. It controls the life cycle of the `modesteditor.core` plugin and thus of the editor that comes along with it.

The class complies to the *singleton* pattern, in this way all classes in the plugin can obtain a reference to this class. This is very helpful since the `ModestEditorPlugin` class is a container for the variables which are needed throughout the whole plugin. One of these variables is the `PLUGIN_ID` which is unique for the plugin, other variables are names of the fields of the preferences page (c.f. 4.5).

The `ModestEditorPlugin` class also manages a collection of references to other singleton classes such as the `ModestTextStyleProvider` and `ModestCodeScanner` which we'll see later. The singleton pattern is realized with a `get` function that tests if the corresponding variable is set, if it isn't, an instance of the class will be created and will accompany the plugin during its whole lifetime.

4.1.2 ModestEditor

The `ModestEditor` class represents the editor itself. Before everything else it's initialized with the function `initializeEditor` which uses the same function from the superclass to get the

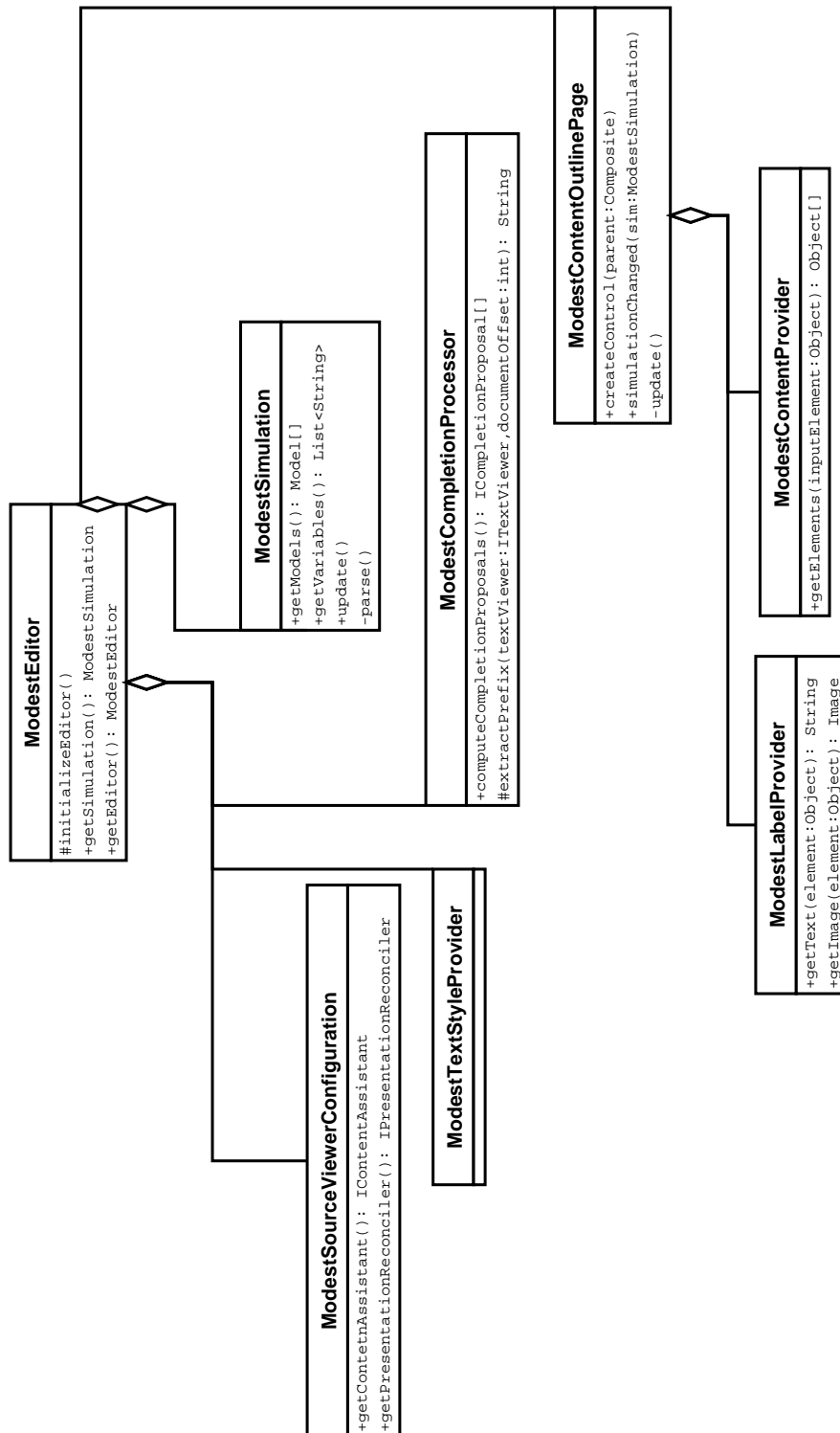


Figure 4.1: The main classes of the modesteditor.core plugin

basics initialized and then sets the `PreferenceStore` where the preferences of the editor are saved, before it sets the `SourceViewerConfiguration` that controls how the entered code is displayed. The editor class is also responsible for supplying its `OutlinePage` to the Eclipse core. This works with the `getAdapter` method which gives back a reference of a singleton `ModestContentOutlinePage`, the `OutlinePage` can be used as singleton since every editor has to have its own `OutlinePage`. The `OutlinePage` will be discussed in more detail in 4.4. Another singleton in the class is a `ModestSimulation` (c.f. 4.4.2) which is a high-level representation of the code entered in the editor and is used a root of the tree displayed in the outline. The `ModestEditor` class registers itself with the `ModestSimulation` as listener in order to get notified when changes occur and thus react to those, for example update the outline.

4.2 Syntax highlighting and content assistance

The `ModestSourceViewerConfiguration` class configures the `SourceViewer` of the `ModestEditor`, that is it tells the editor how to display the entered text and what actions can be taken out of the editing field. The class can be used to supply syntax highlighting and content assistance in order to simplify the work of the developer.

4.2.1 Syntax highlighting

Syntax highlighting is done on hand of a `PresentationReconciler` which are obtained from the `ModestSourceViewerConfiguration` with the method `getPresentationReconciler` which returns `PresentationReconciler` for single line, multi line comments and the other text regions.

The `PresentationReconciler` has `DefaultDamagerRepairer` which supply the information how to display the text. These `DamagerRepairer` get their information, namely `TextAttributes`, from the `ModestTextStyleProvider`. The `StyleProvider` gives the font style and color for every text region. The colors can be adapted in the preferences (c.f. 4.5).

The single line and multi line comment regions are easily recognized, for the keywords which shall also be part of the syntax highlighting one needs a special token scanner in order to properly recognize the different kinds of keywords. This is done via the `ModestCodeScanner` (contained in the package `modesteditor.core.modest`). The `CodeScanner` has rules (see *org.eclipse.jface.text.rules.IRule*) for the different kinds of keywords. Every rule is associated to an `IToken` (see *org.eclipse.jface.text.rules.IToken*) containing the appropriate `TextStyle` obtained from the `ModestTextStyleProvider`.

4.2.2 Content assistance

For the time being the content assistance is completely realized in the `ModestCompletionProcessor` class. `computeCompletionProposals` is hereby the main method, in which the proposals that are seen in the pop-up are actually computed. On hand of the method `extractPrefix`

the word prefix is extracted and the compute method tries to match the prefix against the list of predefined keywords. If a prefix matches the keyword is put in the result array of proposals. Once this is done the list of variables is retrieved from the `ModestSimulation` and the compute method tries to match the prefix against the variables and if the match of a variable is positive this variable is put into the proposals list which will be displayed to the user.

4.3 Error markup

The Eclipse framework offers the programmer some nice ways to display information for the user in the editor. This is done with `Marker` that are registered with resources of the workspace. This means that error markup **cannot** be done for files outside of the current workspace.

In order to place these Markers correctly so called `Regions` are used to determine the placement. In the plugin there are two different categories of markup. The first is a markup for syntactical errors recognized by the parser. The second is a markup for unused variables that are declared but not used in the further program code and thus superfluous. The syntax error `Regions` are computed in the parser and the unused variables `Regions` are computed in the `createWarnings` method of the `ModestSimulation` class during each parse run.

The `ModestEditor` class finally creates the `Marker` from the list of errors and unused variables with the methods `addErrorMarkers` and `addWarningMarkers`. The created `Marker` are only different by their *Severity* that is for the errors `ERROR` and for the unused variables `WARNING`. Those Markers are then represented (in default behaviour) by a cross in a red circle in the left ruler of the editor and red squiggles under the corresponding text for the error Markers and the warning Markers are represented by a yellow triangle in the left ruler and by yellow squiggles in the text.

4.4 Outline

Many Eclipse plugins which handle structured program code come with an outline. The outline view [9] gives a quick overview over the structure of the entered code. On hand of this overview the developer doesn't lose track of the processes and variables easily and can always have a quick look at the outline as a reference.

The parsing and model building was already discussed in 3. So this chapter will only illustrate how the outline is built from an existing tree model. The start makes the central class of the outline.

4.4.1 ModestContentOutlinePage

The `ModestContentOutlinePage` class is the class which controls the display of the outline corresponding to an editor. In the build process we leaned on the article about *TreeView* [11] in the Eclipse knowledge database. In the constructor we register the `ContentOutlinePage` as

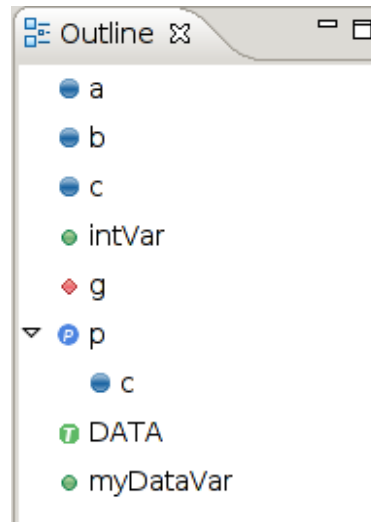


Figure 4.2: The outline View

a listener in the `ModestSimulation` of the editor in order to get notified when the simulation changes. In the `createControl` method, which is called during the initialization, we set up a `TreeView` to display the content, then set the `ContentProvider` to a `ModestContentProvider` and the `LabelProvider` to a `ModestLabelProvider` and finally we set the input of the `TreeView` to the `ModestSimulation` of the editor. The `ContentProvider` is responsible for the navigation through the model tree, each method gets a `Model` (c.f. 4.4.2) and calls the corresponding method of the `Model`. The `LabelProvider` is responsible for the labelling of the tree in the `TreeView`, this includes the images and the text of the nodes.

In order to keep the `OutlinePage` up to date, it registered itself as a listener to the `ModestSimulation` so that the `ModestSimulation` is responsible for the updating whenever it is modified. When an update happens the `simulationChanged` method is called which in turn calls the `update` method. The `update` method refreshes the `TreeView`.

Every outline view disposes of a selection mechanism. When the user selects something in the outline it is custom in Eclipse to highlight the declaration of the selection and to direct the cursor to this position. This is done in the `selectionChanged` method. This method retrieves the position informations from the selected `Model` and with the `selectAndReveal` method from the editor gets the highlighting done.

4.4.2 Models

The models in the tree of the `TreeView` are all subclasses of the `Model` class which provides a common framework for all models, such as a name and a parent `Model` which is either null if the `Model` is at the top level or a process if the declaration is in a process declaration.

ModestSimulation

The `ModestSimulation` class is a container for the `Models` that are extracted from the document. The simulation class follows a singleton pattern and is responsible for itself. That means that it has to rebuild the `Models` on an update of the document and after that notify the registered listeners that it has changed. The simulation class has methods to add and remove those listeners and has methods to return the contained `Models` to the caller. To get the update procedure going there is an `update` method that calls the `parse` method (note that here occurs the only parsing for the outline). The class is also the store for the current AST that can be put to further use in other parts of the plugin.

Process

The `Process` class stands for a process declaration in the entered document. It is a subclass of the `Model` class. In addition to the inherited attributes and methods it has to take care of the hierarchy. A process is namely the only `MoDeST` construct which can have declarations in it. Thus it is the only `Model` that has children in the outline tree. The class therefore has methods to add and remove children and to get them for further work such as displaying in the outline.

Others

The other `Model` classes only add very small functionality to the base class. The subclassing is needed in order to have attributes only used in this construct and not an excess that doesn't fit the actual declaration. So for example the `ModestAction` class has an attribute `patient` that reveals whether the action is patient or not in the `MoDeST` context.

4.5 Editor Preferences

Typically an user wants to be able to customize some things in his editor. In order not to have to recompile the whole editor, preferences were introduced. So an user can easily change some of the behaviour of the editor. Some kind of preferences are already defined by Eclipse (such as the font settings), others were newly introduced. In the main preference page the user can only change the colors of the syntax highlighting, later on there are some subpages to come.

4.5.1 Default initialization

The default initialization of the main page happens on hand of the method `initializeDefaultPluginPreferences` in the `ModestEditorPlugin` class. The `initialize` method is called in the `start` method of the class in order to set the defaults for the self defined preferences.

All preferences are maintained in a unique store for the plugin and referenced through unique names. These names are known through string variables declared in the plugin class, so they can be accessed from every class in the plugin since the plugin class is a singleton.

4.5.2 ModestPreferencePage

The preferences of the plugin are managed in the `ModestPreferencePage` which is an extension of the Eclipse class `FieldEditorPreferencePage`. First it is needed to set the used preference store, which we get from the `ModestEditorPlugin` class. After the `createFieldEditors` method is called which creates five `ColorFieldEditor` for the five newly defined colors. The rest is managed through the preference framework of Eclipse. Further it is possible to disable the error markup by a checkbox. In the default the error markup is enabled.

5 Launch framework integration

In the previous chapter, the means to easily enter and alter MoDeST code were shown. In this chapter, the possibilities to use this previously specified code will be presented. Since MoDeST has an already existing toolchain with a compiler and an own simulator it was only natural to integrate the existing tools into the new plugin for Eclipse. In order to get this integration a new plugin was created with the name `modesteditor.launch`. The launch part points out the connection with the Eclipse launch framework [17]. This launch framework was designed to give plugin developers a platform for the integration of external tools into Eclipse and the developed plugins.

The goal in the MoDeST plugin was not only to integrate the MoDeST compiler and its possible dot output but also to give access to the FSNS (*First State Next State*) interface. FSNS is a state simulator that simulates MoDeST specifications with a textual user interface. In order to keep the execution of the tools simple a preference page for the launch plugin was created which comprises the relevant paths.

First the main classes of the `modesteditor.launch` plugin, such as the startup class and the delegate class which starts the actual programs. Then the preferences of the plugin will be shown, divided into the preference page and the launch tab window which holds the informations for the program arguments.

5.1 The main classes

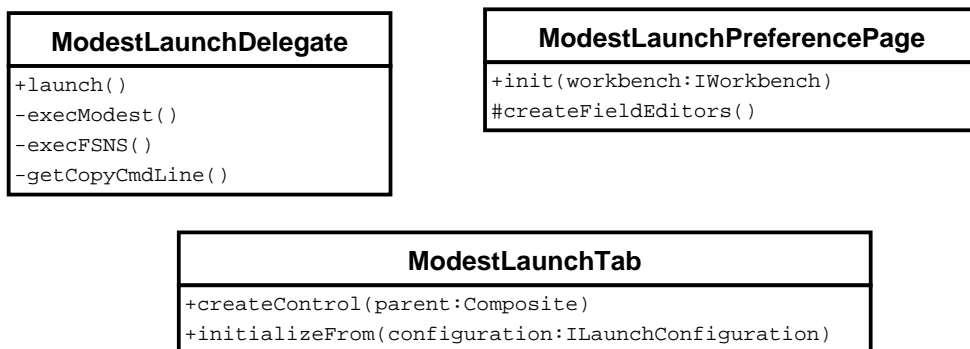


Figure 5.1: The main classes of the `modesteditor.launch` plugin

5.1.1 ModestLaunchPlugin

The activator class of the `modesteditor.launch` plugin is the `ModestLaunchPlugin` class. As seen before, it follows a *singleton* pattern and can be accessed through the `getDefault` method. The class is also a container for the string variables that represent the names of the plugin specific preferences such as `COMPILER_LOCATION`. In the `start` method the default preferences (c.f. 5.2) are set.

5.1.2 ModestLaunchDelegate

The `ModestLaunchDelegate` class implements the `ILaunchConfigurationDelegate` interface which is part of the Eclipse launch framework. The interface only requires a `launch` method that takes a `LaunchConfiguration` and executes an `IProcess`. The `LaunchConfiguration` is obtained from the run dialog of Eclipse. This configuration combined with the preferences gives all the information needed to create a process that compiles the document from the editor.

The actual `IProcess` creation for a compilation is done in the `execModest` method. After having executed the `IProcess`, the obtained files are copied to the specified output location on hand of the information provided by the `getCopyCmdLine` method. To copy the files the Jakarta commons IO package [12] is used because it provides easy file manipulation without having to manage low level Java readers.

If the FSNS checkbox is checked in the `LaunchTab` the `fsns` binary is launched, not the compiler. The simulation is displayed in the Eclipse `Console` view and the user choices are entered there too. How this looks like can be seen in figure 2.8.

5.2 Launch Preferences

5.2.1 ModestLaunchPreferencePage

As we've seen in the main preference page (c.f. 4.5) the `PreferencePage` for the launch plugin is a `FieldEditorPreferencePage`. In the extension point definition (c.f. 7.2.1) the `ModestLaunchPreferencePage` is put in the same category as the main preference page for MoDeST, thus it is displayed as a subpage of the main page and all MoDeST specific preferences are summarized under one menu item to simplify the finding for the user.

In the constructor we set the `PreferenceStore` to the `PreferenceStore` of the plugin. In this `PreferenceStore` on hand of the `createFieldEditors` method we create fields for the path to the compiler, the path to the FSNS binary, the path to the output directory and a checkbox for the dot file creation.

5.2.2 ModestLaunchTab

In order to really be able to call a MoDeST program through the run dialog of Eclipse, it is necessary to define a `LaunchTab` (c.f. 7.2.3). This `LaunchTab` provides Eclipse with the `Launch-`

Configuration previously mentioned and holds the information for the arguments of the programs called by Eclipse. In order to run the compiler not much is needed. So we just have to enter the project name and the file name of the file to compile. In addition to this, the user is permitted to select if he wants a dot file as output. This flag is set if the corresponding flag is set in the preferences.

If the “run FSNS” checkbox is checked the fsns binary is launched rather than the compiler. In this way the user can access a FSNS step simulation in the Eclipse Console view.

6 Step simulation

After having seen in chapter 2 and the previous chapter how the MoDeST plugin can be used to simulate a MoDeST specification with FSNS, this chapter will deal with a new way to simulate MoDeST code. As seen in figure 2.8, the output of the FSNS interface isn't that expressive, especially because there is no real reference to the code used for the simulation. This combined with the wish to be able to test out the behaviour of some MoDeST code with an immediate feedback was the reason why a new simulation was created. This step simulation was to be fully integrated into the Eclipse framework and the MoDeST plugin and thus could dispose of a direct access to the code in the MoDeST editor of the plugin. This allowed visual highlighting of the active parts in the selected transitions.

The realisation of the step simulation was done incrementally beginning with a subset of the MoDeST grammar called the MoDeST core. In a first step, process definitions, calls and the exception handling were added to the core. Next the relabeling constructs were put in before adding probabilistic transitions, guards and assignments to the language set. After this last step the full MoDeST language was supported.

For a better grasp of the theory model behind the simulation a certain familiarity with the MoDeST semantics is needed, therefore please refer to 2.4.1. This theory model is presented first in the afore mentioned incremental fashion. Then a discussion of the simulation implementation is shown starting with the Eclipse integration before getting to the theory implementation and finishing with the extra features.

6.1 The simulation theory

An introduction to the theory model of the step simulation will be given next. This introduction will present the relation between the model and the operational semantics of MoDeST. This semantics as it is explained in [5] is an action driven one. This means that a step in the state automaton is triggered by a MoDeST action. Such a transition looks like the figure 6.1.

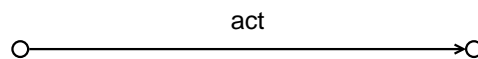


Figure 6.1: A transition in the MoDeST core

When one looks at the tree representation of a MoDeST program as it is given by the MoDeST process behaviour grammar, one will see that the actions are always placed at the leaves of

the tree. The only other MoDeST language construct placed at the leaf position is the `throw` statement, but this will be discussed later.

So a transition is basically the same as a consumption of a leaf node of the tree. What happens with the rest of the tree when the leaf was consumed depends on the parent node of the consumed leaf. In order to get a better notion of this consumption model, it will be discussed in the following giving the theory model incrementally beginning with a small subset. The exact semantic rules will not be presented in detail since they can be appreciated in [5].

6.1.1 The MoDeST core

To begin with, only a subset of the MoDeST grammar will be inspected. This subset was called MoDeST core by Prof. Hermanns in one of his lectures. The subset is the following:

$MoDeST_{core} = \{P ::= act \mid P_1; P_2 \mid \text{alt}\{:: P_1 \dots :: P_k\} \mid \text{do}\{:: P_1 \dots :: P_k\} \mid \text{par}\{:: P_1 \dots :: P_k\}\}$ with P denoting a process and act and action. Actions can be self-declared or the built-in actions `STOP`, `tau` and `break`.

Such a process P can be one of the following:

- a sole action. The representing tree is formed only by one node namely the action itself. When it is consumed the whole tree is consumed and the simulation is terminated successfully.
- a sequential process. This process is the root node of the tree and the first child is P_1 and the second child is P_2 . When the first child is successfully terminated, the second child will be inspected. When this one is terminated too, the whole tree is terminated.
- an `alt` construct. This construct is the root of the tree and its children are alternatives. This means that only one of the alternatives can be active at a time and when one of these subtrees are terminated the whole tree is terminated.
- a `do` construct. This construct behaves very like the `alt` construct. The difference is that when a subtree is terminated the whole tree returns to its original state. A special case is a `break` action that will be seen before long.
- a `par` construct. As with the `alt` construct, the `par` construct is the root of the tree and has alternatives as children. Here all children are executed in parallel as the name says. As seen in [5] the execution of the children is synchronized over the action in the common alphabet. An action is in the common alphabet of a `par` construct if the action occurs in another alternative of the `par` construct. The synchronization rules says that all actions in the common alphabet have to be executed in parallel, note that this synchronization is blocking. Remember the common alphabet of a `par` construct is built at the beginning of the program execution and remains the same until the end of the execution. Thus an action has to synchronize over the common alphabet even if the action was already taken once. The tree terminates when all children are terminated.

Two special cases of an action can occur in the MoDeST core. The first one is a **break** action that creates a signal on consumption. This signal goes up in the tree hierarchy until it meets a **do** node or the root. In the case it meets a **do** node, this node is terminated. In the other case only the action node is terminated and the tree behaves as if a normal action was taken. The second special case is a **STOP** action. If this action is consumed the tree stops every further execution and terminates but not successfully!

6.1.2 Processes and exception handling

The MoDeST behaviour grammar is now extended by process calls and exception handling. The syntax is then:

$$MoDeST_{proc} = MoDeST_{core} \cup \{ProcName(e_1, \dots, e_k) \mid \mathbf{throw}(exc_p) \mid \mathbf{try}\{P\} \mathbf{catch} \ exc_p_1 \{P_1\} \dots \mathbf{catch} \ exc_p_k \{P_k\}\}$$

A process call is a substitution of the call by the body with the variables instantiated from the values of the arguments in the call. MoDeST disposes of an exception handling mechanism very similar to the one known from Java. An exception can be thrown via a **throw** statement and thus interrupts the normal process execution. If it is not explicitly caught by a **catch** statement, it terminates the program execution. If the exception is caught the process declared in the **catch** statement is executed next.

6.1.3 Relabeling

The MoDeST behaviour grammar is now extended by the relabeling language features. The syntax is then:

$$MoDeST_{label} = MoDeST_{proc} \cup \{P ::= \mathbf{hide}\{act_1, \dots, act_k\} \mid \mathbf{extend}\{act_1, \dots, act_k\} \mid \mathbf{relabel}\{a_1, \dots, a_k\} \mathbf{by} \{a'_1, \dots, a'_k\}\}$$

The synchronization in a parallel setting is done over the actions in the common alphabet. In order to modify this synchronization three language constructs were introduced: **hide**, **extend** and **relabel**. These constructs modify the process declared after the the respective construct.

- The **hide** construct serves, as it's name already suggests, to hide actions before the context. The actions in the **hide** statement are not subject to synchronization in parallel processes even if they are in the common alphabet.
- The **extend** construct extends the alphabet of a process with the actions occurring in the **extend** statement. This has no impact for the behaviour of the process following the statement only to the behaviour of the parallel process enclosing the statement.

- The `relabel` construct is a renaming of the actions before and after the `by` keyword. The action with the index k of the list before the `by` is renamed in the action with the index k of the list after the `by` keyword. The alphabet of the process following the `relabel` statement is modified accordingly.

6.1.4 `palt`, guards and assignments

The MoDeST behaviour grammar is now extended by probabilistic alternatives, guards and assignments. The syntax is then:

$$\begin{aligned} \text{MoDeST} &= \text{MoDeST}_{\text{label}} \cup \{P ::= \text{when}(b) P \mid \text{urgent}(b) P \mid \\ &\quad \text{act palt} \{ : w_1 : \text{asgn}_1; P_1 \dots : w_k : \text{asgn}_k; P_k \} \} \end{aligned}$$

Where b is an expression denoting a constraint, w_i is a weight for the alternative and asgn_i is an assignment block. The weights are used to determine the probability of an alternative.

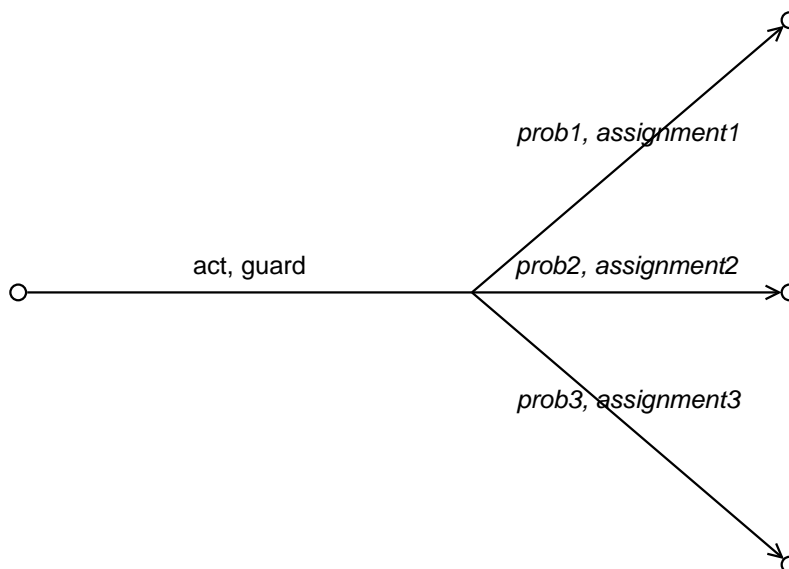


Figure 6.2: A probabilistic transition

With this extension the semantics is extended by a new type of transition, so called probabilistic transitions. An example is depicted in 6.2. The picture shows how the transition should be handled, first the action `act` is taken and then a probabilistic choice is done and the corresponding assignments are executed in an atomic way.

In the case the action `act` of the probabilistic transition is combined with others in a parallel transitions the probabilistic choice has still to be executed in the same way. If there are more than one of those “`palt`”-transitions with the same action they are combined too, see 6.3.

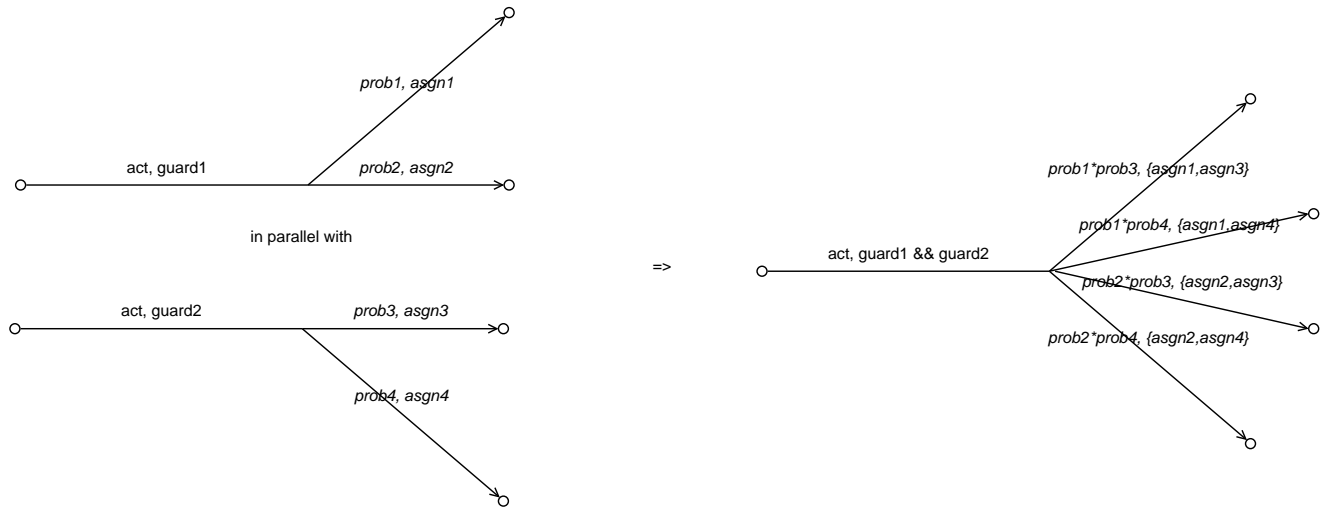


Figure 6.3: Parallel probabilistic transitions over the same action act

6.2 The simulation implementation

After having seen the theoretic model of the step simulation, the implementation of this model will be discussed next. This implementation is done in the `modesteditor.core.stepsimulation` package. All mentioned classes in this section are to be looked for in this package. For simplicity the naming of the classes was kept close to the name of the corresponding MoDeST constructs. The different classes related to the theory model and their methods will be inspected in the same incremental setup as the theory for a better understanding. First the framework of the simulation which integrates the simulation into Eclipse is shown. Next the implementation of the theory model is discussed in detail before going over to the backtracking feature as an extra.

6.2.1 The framework

The framework consists of the `SimulationAction`, the `SimulationRoot`, the model tree and the `SimulationView` class. The `SimulationAction` is responsible for the startup of the simulation. It has to check if a *simulation View* is already opened and after that starts the `TreeWalker` (c.f. 3.3) to obtain the model tree. The `SimulationRoot` class is the root element of the `TreeView` used in the `SimulationView`. It holds the root of the model tree and provides the interface for the communication with the Viewer. The model tree is built of `SimulationNode` instances and is very close to the parse tree of the MoDeST grammar.

The abstract class `SimulationNode` is the parent class of all classes representing the MoDeST language constructs. This class regroups all fundamental methods and attributes every construct should have. The basic methods are implemented and the method that must be differentiated

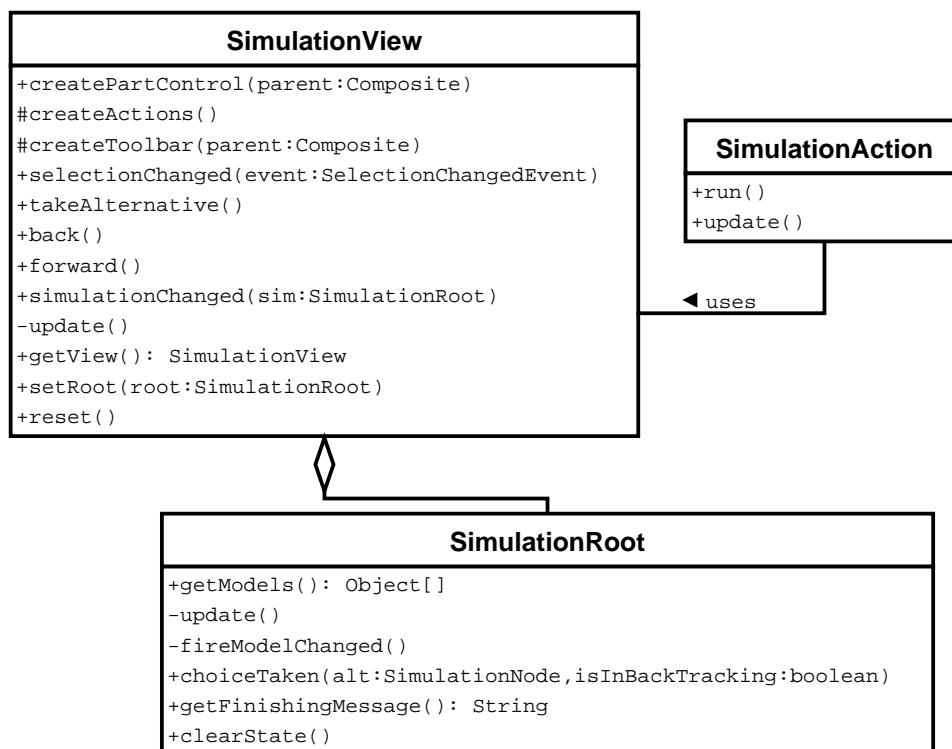


Figure 6.4: The simulation framework classes

within the representing language construct are left abstract. Methods as `collectTransitions` or `collectParTransitions` are left abstract since every construct has other ways to take transitions.

The `SimulationView` class represents the View used for the interaction with the user during the step simulation. The class follows a singleton pattern in order to easily have access to the View from other classes such as the `SimulationAction`. The content of the View are the possible transitions that are presented on hand of a `TreeView` [11]. The View has one action for choosing the selected transition and two actions for the backtracking (c.f. 6.2.6).

6.2.2 The MoDeST core

In this part the core of the MoDeST language as it is presented in 6.1.1 is discussed.

Actions

The `ActionNode` class represents all MoDeST actions except `break` that has an own representation. The `ActionNode` has as alphabet it's name and gives back itself when the `collectTransitions` method is called. When the action is then taken the instance is set non active and notifies the parent that it is finished by calling the `childIsFinished` method.

alt

The `AltNode` class represents an `alt` construct in MoDeST as children it has instances of the `Alternative` class. Those instances are only container for the underlying sequential processes that give everything through to them. The `AltNode` collects the transitions from all the children in a list and gives this list back so that the user can choose from this list.

do

The `DoNode` class is a subclass of the `AltNode` since it behaves mostly similar. A difference is that when a child signals that it is finished via the `childIsFinished` method, the `DoNode` doesn't finish but sets active all children again. This is done until a `break` is taken and the `DoNode` gets the notification via the `breakTransitionTaken` method then all child nodes are set inactive on hand of the `releaseTransitions` method and the `DoNode` is finished too.

break

The `BreakNode` class represents a `break` in the MoDeST language and behaves very much like an `ActionNode`. But the `BreakNode` has no alphabet and triggers the `breakTransitionTaken` method when it is taken.

Sequential Processes

The `SequentialProcess` class represents processes that are executed sequentially and are expressed by a separation via “;” in MoDeST. Since the children are to be executed sequentially only the first child in the list is set active on start. When this child is finished the next is set active and when the last is finished the process is finished too.

par

The `ParNode` class represents the `par` construct in the MoDeST language. The class builds the common alphabet when set active. This alphabet are the actions that occur in pairwise distinct children of the `par` construct. The `ParNode` is responsible for collecting the parallel transitions that are taken over the common alphabet. Thus when the `collectTransitions` method is called, it calls the `collectParTransitions` method. This method is responsible for putting the parallel actions into one `ParTransition` instance and to check if all of these actions are present in the other case the transition is disabled. When all children are finished the node is finished too.

Parallel transitions

The `ParTransition` class is the representation of a parallel transition and thus a container for actions within the common alphabet with the same name. The transition is only enabled when all actions contained are enabled too. When the transition is taken all the contained actions are taken via a call of the `takeTransition` method of the `ActionNode`. Since a call of `collectParTransitions` always creates instances of the `ParTransition` class, it has to be ensured that `break`, `STOP` and `tau` actions are always enabled since they are never in the common alphabet.

6.2.3 Processes and exception handling

The now discussed addition to the core are seen in 6.1.2. Process definitions and calls are already discussed in 3.3 since they are handled completely by the parsing framework. A process call is represented as a normal program continuation and there is no need for a special handling.

The exception handling in the simulation is done on hand of two specific classes. The first one is the `ThrowNode` representing a `throw` statement in the MoDeST language. When this throw-statement is taken in a transition the `exceptionThrown` method of the `SimulationNode` is called. This method propagates the exception to the root of the tree until it reaches a `TryNode` or the root of the tree. If the exception reaches the root of the tree the simulation is aborted with an according message.

The `TryNode` class represents a `try` block with the related `catch` blocks in the MoDeST language. This class behaves as a container for the code inside the try-block until an exception is thrown inside the block. Then the `exceptionThron` method is called and in this method a

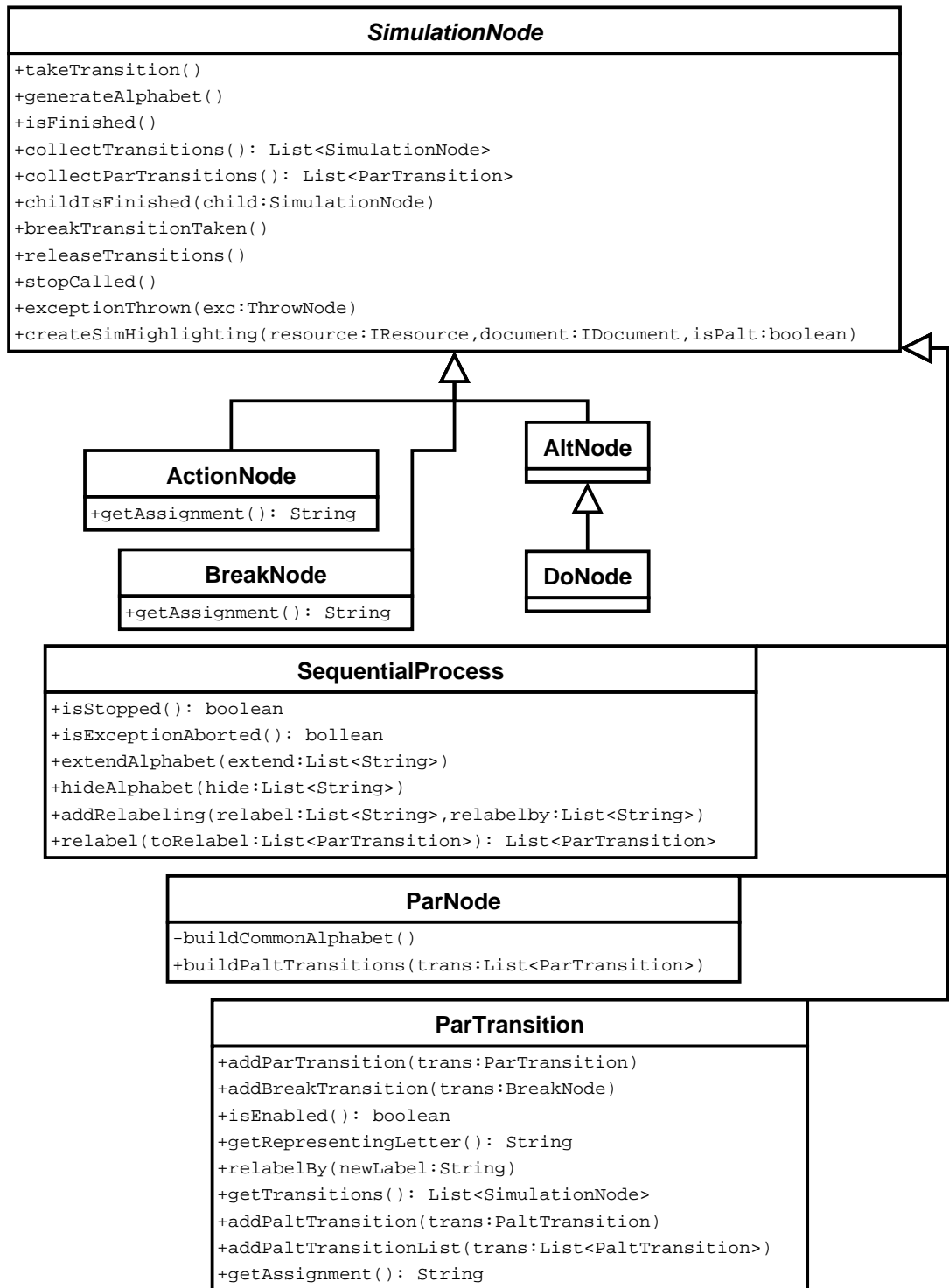


Figure 6.5: The classes of the MoDeST core

lookup occurs that checks whether a corresponding catch statement is available or not. If so the code of the catch statement is executed next, if not the `exceptionThrown` method of the parent is called in order to see if there is another surrounding try-block with a adequate catch statement. The alphabet of the node is built from the alphabet of the try-block and of the alphabets of all catch-blocks. During the build of the tree the catch statements are added with the `addCatch` method which builds the map from exception to the code held in the statement in order to later know what code to execute for a specified exception.

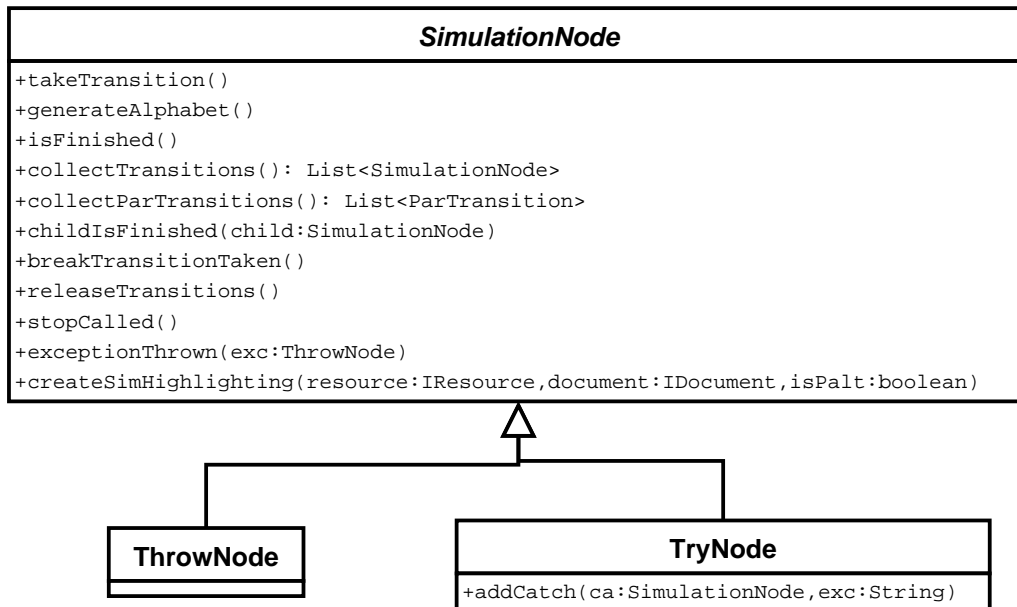


Figure 6.6: The additional classes for the exception handling

6.2.4 Relabeling

The next addition to the MoDeST language is the one presented in 6.1.3. The three relabeling constructs are always placed before a sequential process that is the scope of the relabeling. For this reason the only class in which anything is done about relabeling is the `SequentialProcess` class.

For the `extend` construct the `extendAlphabet` method is called which adds the actions in the `extend` statement to list which is by turn added to the alphabet in the `generateAlphabet` method. This results in disabled parallel transitions if the process doesn't generate transitions with these actions.

For the `hide` construct the `hideAlphabet` method is called which saves the actions in the `hide` statement in a list which is then removed from the alphabet in the `generateAlphabet` method. Since these actions are not in the alphabet anymore the synchronization ignores them.

For the `relabel` construct the `addRelabeling` method is called which generates a mapping from the old actions to the new ones. In the `generateAlphabet` method this mapping is applied to the generated alphabet. In the `collectParTransitions` method the transitions collected are relabeled by a call of the `relabel` method which applies the mapping to the transitions.

6.2.5 `palt`, guards and assignments

The last addition to complete the MoDeST language is the one presented in 6.1.4. Since guards can be placed in front of every process a guard attribute was added in the `SimulationNode` class and is set in the constructor during the pass of the tree walker (c.f. 3.3) and presented to the user with the transitions.

Assignments occur either in conjunction with a MoDeST action or in the context of a `palt` construct. Therefore an assignment attribute was placed in the `ActionNode` class to offer a representation of the assignments during the transition presentation. Assignments occurring in a `palt` construct are handled in the `PaltTransition` classes discussed next with the representation of the `palt` construct.

This representation is implemented in the `PaltNode` class. The alphabet is generated from the action guarding the `palt` (when present) and from all weighed alternatives. When collecting transitions in a non parallel context the `PaltNode` instance returns itself and when taken it calls the `takeTransition` method of the action guard that is presented beforehand to the user as action of the transition. This done the probabilistic transitions are presented to the user after being obtained with the `getPaltTransitions` method. These `PaltTransitions` now display their respective probability and the assignment to occur when taken. These probabilistic transitions are to be taken before any other transition since they have to happen in an atomic way (c.f. 6.2). When transitions are collected in a parallel context the `PaltTransitions` are added to the created `ParTransition` with the `addPaltTransitionList` of the `ParTransition` class. Here too the atomicity has to be respected, thus when two `ParTransitions` have probabilistic transitions, those transitions are merged in a multiplicative way as shown in 6.3. This can lead to a deadlock since the twigs are all presented to the user as the MoToR simulator would also choose the twig in a probabilistic fashion, without looking forward for enabledness of the next action to be chosen.

6.2.6 Backtracking

As an extra a backtracking system was implemented for the simulation. This gives the opportunity to go back one or more steps by simply clicking a button with the mouse. This system not only allows to go back some steps, but also to go forward again and thus to redo some of the undone steps.

Since the simulation doesn't build on a state automaton model but is based on a consumption model it is hard to go back to a previous state. The only state easily obtained is the starting state. By remembering the steps taken one can access a previous state by redoing some of the

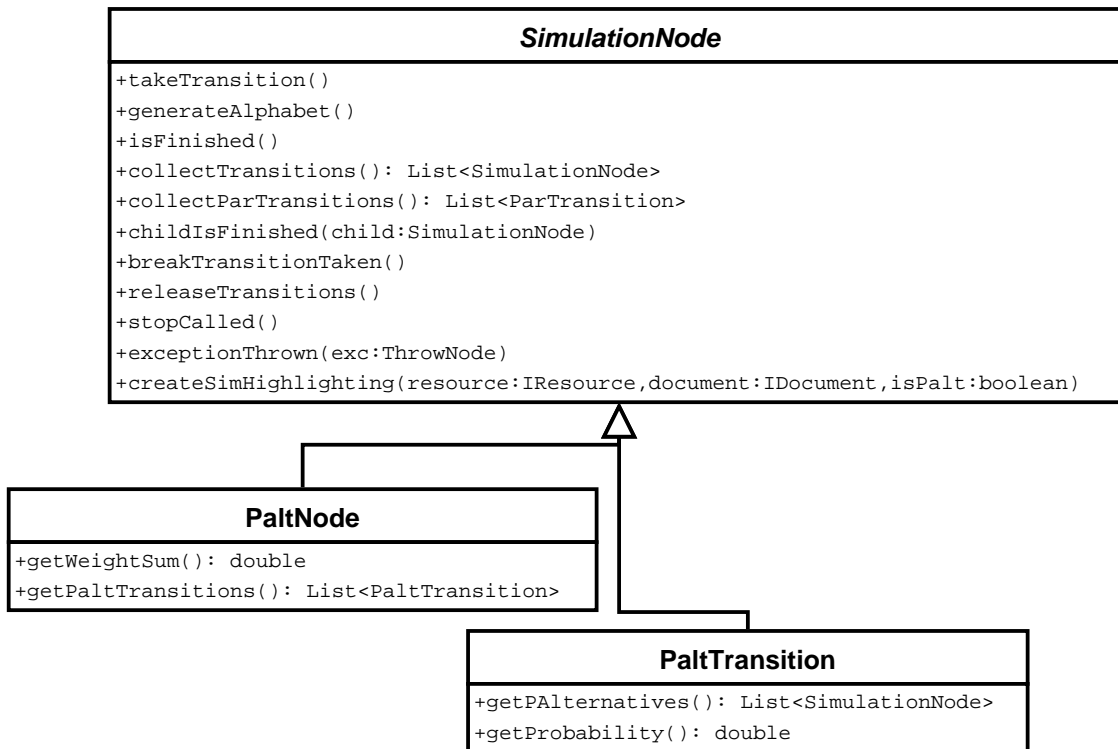


Figure 6.7: The additional classes for handling probabilistic transitions

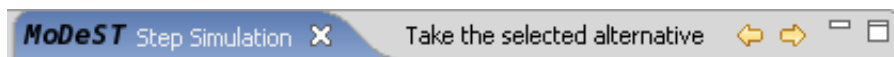


Figure 6.8: The step simulation View with the backtracking arrows

steps beginning at the starting state. Therefore all transitions taken are stored in a list in the `SimulationView` class in an attribute called `choiceStack` that holds all transitions taken so far in the respective order. When the back or forward actions are called by clicking the arrows in the simulation View, the `back` or `forward` methods of the `SimulationView` class manage the actual state reset and the restoring of the looked for state by redoing the transitions saved in the `choiceStack`. Here should be mentioned that the probabilistic transitions are atomic and the probabilistic choice of one of these transitions can't be undone alone.

7 Extension points used

After the presentation of the complete implementation, this chapter will present the extension points used during the design of the MoDeST plugin. These extension points [7, 8] are the predefined entry points of new functionality for Eclipse. Since Eclipse is completely plugin based a concept was needed in order to get the different plugins working together. Extension points build the base of this working together. They are defined in the *plugin.xml* file of the plugin which offers a functionality corresponding to a specified extension point. This is the way (and the only one) Eclipse knows how a plugin is supposed to extend the already existing functionality of Eclipse. This is also why the *plugin.xml* file plays such a crucial role in the interaction between a plugin and Eclipse.

In order to simplify the access to the MoDeST plugin, in the following a reference of the used extension points is shown. First the ones for the `modesteditor.core` plugin are presented before coming to the points of the `modesteditor.launch` plugin.

7.1 modesteditor.core

7.1.1 org.eclipse.ui.editors

The `org.eclipse.ui.editors` extension point registers a new editor for an Eclipse file object. In this way Eclipse knows that it has to start the custom editor when editing a file with the specified extension. In the following the definition of the extension point is presented:

```
<extension
  point="org.eclipse.ui.editors">
  <editor
    name="Modest Editor"
    icon="icons/file.gif"
    extensions="modest"
    contributorClass="modesteditor.core.ModestActionContributor"
    class="modesteditor.core.ModestEditor"
    id="ModestEditor">
  </editor>
</extension>
```

Here with the `id` *ModestEditor* and the implementing class *modesteditor.core.ModestEditor*. The Editor support files with the following extensions: *modest*. The name of the editor is

specified in the name tag as seen above.

7.1.2 org.eclipse.core.filebuffers.documentSetup

The `org.eclipse.core.filebuffers.documentSetup` extension point registers a document setup participant that is involved during the initialization process of a text file buffer for a file with the specified extension. In the following the definition of the extension point is presented:

```
<extension
    id="ModestDocumentSetupParticipant"
    name="Modest plugin documentSetupParticipant"
    point="org.eclipse.core.filebuffers.documentSetup">
    <participant
        extensions="modest"
        class="modesteditor.core.ModestDocumentSetupParticipant">
    </participant>
</extension>
```

The class `modesteditor.core.ModestDocumentSetupParticipant` is registered as a setup participant for the file extension `modest`.

7.1.3 org.eclipse.ui.preferencePages

The `org.eclipse.ui.preferencePages` extension point allows to register new preference pages in the common preference dialog box of Eclipse. It further allows to specify a name for the page which appears in the list of preferences and a category if there should be more than one page for some subject. In the following the definition of the extension point is presented:

```
<extension
    point="org.eclipse.ui.preferencePages">
    <page
        name="MoDeST"
        class="modesteditor.core.preferences.ModestEditorPreferencePage"
        id="modesteditor.core.ModestEditorPreferencePage">
    </page>
</extension>
```

The name of the preference page is given as `MoDeST`. The implementing class is defined as `modesteditor.core.preferences.ModestEditorPreferencePage` and the `id` is set to `modesteditor.core.ModestEditorPreferencePage`, this will be significant later on, when other preference pages are set to be subpages of the main preference page defined with this extension point.

7.1.4 org.eclipse.ui.views

The `org.eclipse.ui.views` extension point allows to register new Views for the Eclipse workbench. In the definition is already specified which name the View will have and also where to find an icon for this View. Further it is possible to define categories if some Views shall be bundled subjectwise. In the following the definition of the extension point is presented:

```
<extension
  point="org.eclipse.ui.views">
  <category
    id="modesteditor.core"
    name="MoDeST">
  </category>
  <view
    id="modesteditor.core.SimulationView"
    name="Step Simulation"
    icon="icons/MoDeST.png"
    category="modesteditor.core"
    class="modesteditor.core.stepsimulation.SimulationView">
  </view>
</extension>
```

First a new category for the Views originating from the MoDeST plugins is defined. The then defined View is given an id and is named *Step Simulation*. The icon of the View is set to *icons/MoDeST.png* and the class representing the View is set to *modesteditor.core.stepsimulation.SimulationView*.

7.2 modesteditor.launch

7.2.1 org.eclipse.ui.preferencePages

This extension point was seen before and is used to register a new preference page for the `modesteditor.launch` plugin. In the following the definition of the extension point is presented:

```
<extension
  point="org.eclipse.ui.preferencePages">
  <page
    name="Run"
    category="modesteditor.core.ModestEditorPreferencePage"
    class="modesteditor.launch.ui.ModestLaunchPreferencePage"
    id="modesteditor.launch.ModestLaunchPreferencePage">
```

```
</page>
</extension>
```

The name of the preference page is set to *Run* to symbolize the launch relevant character of the preference page. The other important definition is the `class` tag, which is set to *modesteditor.core.ModestEditorPreferencePage*. In this way the defined preference page will be a subpage of of the main *MoDeST* preference page that was defined earlier.

7.2.2 org.eclipse.debug.core.launchConfigurationTypes

The `org.eclipse.debug.core.launchConfigurationTypes` extension point provides a configurable mechanism for launching applications. In the definition the modes in which the configurationType can be run are specified and it is given a name. In the following the definition of the extension point is presented:

```
<extension
  point="org.eclipse.debug.core.launchConfigurationTypes">
  <launchConfigurationType
    delegate="modesteditor.launch.ModestLaunchDelegate"
    id="modesteditor.launch.ModestLaunchType"
    modes="run"
    name="Modest"/>
</extension>
```

In this `LaunchConfigurationType` only the modes *run* are defined. For this configuration type the delegate *modesteditor.launch.ModestLaunchDelegate* is set, so that this class will be used to run this configuration type.

7.2.3 org.eclipse.debug.ui.launchConfigurationTabGroups

The `org.eclipse.debug.ui.launchConfigurationTabGroups` extension point provides a mechanism for contributing a group of tabs to the launch configuration dialog for a type of launch configuration. This dialog is where the actual arguments for the starting of the external programs is put. It also manages to some extend the options of the external programs. In the following the definition of the extension point is presented:

```
<extension
  point="org.eclipse.debug.ui.launchConfigurationTabGroups">
  <launchConfigurationTabGroup
    class="modesteditor.launch.ui.ModestLaunchTabGroup"
    id="modesteditor.launch.ui.ModestLaunchTabGroup"
    type="modesteditor.launch.ModestLaunchType"/>
</extension>
```


The tab group is specified in the class *modesteditor.launch.ui.ModestLaunchTabGroup*. It is applicable for the previously defined type *modesteditor.launch.ModestLaunchType*.

7.2.4 **org.eclipse.debug.ui.launchConfigurationTypeImages**

The `org.eclipse.debug.ui.launchConfigurationTypeImages` extension point provides a way to associate an image with a launch configuration type. In the following the definition of the extension point is presented:

```
<extension
  point="org.eclipse.debug.ui.launchConfigurationTypeImages">
  <launchConfigurationTypeImage
    configTypeID="modesteditor.launch.ModestLaunchType"
    icon="icons/launch.gif"
    id="modesteditor.launch.ModestLaunchTypeImage"/>
</extension>
```

The image associated to the already defined configuration type *modesteditor.launch.ModestLaunchType* is set in the `icon` tag to *icons/launch.gif*.

8 Conclusion

After the short introduction, a guide of the plugin features was presented. This guide shall help to give all users of the plugin and the readers of this thesis an easier access to MoDeST and the plugin. In the manual many pictures were used to ease the recognition of the handled concepts and features during the work with the plugin. Next a short overview of the parsing system used in the plugin was given before showing how this system fits into the editor of the plugin. Not only parsing was presented in the editor chapter but also the error checking features and how the editor eases the work for programmers were pictured. In the next chapter the integration of the plugin in the launch framework was presented, this shall help to use the existing MoDeST tools together with Eclipse since not all tools were reproduced in the Eclipse context. Thereafter the key feature of the plugin, the step simulation of MoDeST was shown in detail. Not only the practical part was analyzed but also the theory standing behind was explained. Something similar already existed with the FSNS interface of the MoToR tool, but the simulation in Eclipse has many advantages for the user. The main advantage is probably that the user actually can see which code the simulation is executing. Finally in order to give a good overview to other Eclipse developers a reference of the used extension points was shown in the last chapter. Some further documentation such as the Javadoc documentation of the code can be found on the website of the plugin [14]. The update site for the installation of the plugin can also be found there. In the further development it should be possible to add some more language specific features to the plugin. The next step could be one of two possibilities. The first one being, the discrete event simulation as it is done in MoToR is implemented in Eclipse. This would facilitate the use of the plugin as all features are bundled in one place. Possibly the new implementation could alleviate some of the drawbacks that exist currently in MoToR such as the lack of urgency constraints. The other possibility would be to create an interface between the Eclipse plugin and the Moebius [15] tool. Since for the time being Moebius is the backend for simulation of the MoToR tool it would be possible with such an interface to use the MoToR simulation out of Eclipse. This way seems feasible since Moebius is also written in Java and could possibly be incorporated into an Eclipse plugin. A possible problem could be that Moebius isn't open source and the task to create the interface could be a non-trivial project.

Bibliography

- [1] ANTLR parser generator homepage: <http://www.antlr.org>.
- [2] ANTLR plugin homepage: <http://antlreclipse.sourceforge.net/>.
- [3] Henrik C. Bohnenkamp, Holger Hermanns, Ric Klaren, Angelika Mader, and Yaroslav S. Usenko. Synthesis and stochastic assessment of schedules for lacquer production. In *QEST*, pages 28–37, 2004.
- [4] Pedro R. D’Argenio, Holger Hermanns, Joost-Pieter Katoen, and Ric Klaren. MoDeST – a modelling and description language for stochastic timed systems. In Luca de Alfaro and Stephen Gilmore, editors, *PAPM-PROBMIV*, volume 2165 of *LNCS*, pages 87–104. Springer, 2001.
- [5] Pedro R. D’Argenio, Holger Hermanns, Joost-Pieter Katoen, and Ric Klaren. MoDeST: A compositional modeling formalism for hard and softly timed systems. 2005.
- [6] Eclipse homepage: <http://www.eclipse.org>.
- [7] Definition of an eclipse extension point:
<http://help.eclipse.org/help32/topic/org.eclipse.platform.doc.isv/guide/arch.htm>.
- [8] Eclipse extension point reference:
<http://help.eclipse.org/help32/topic/org.eclipse.platform.doc.isv/reference/extension-points/index.html>.
- [9] Definition of an eclipse outline view:
<http://help.eclipse.org/help32/org.eclipse.platform.doc.user/concepts/coutline.htm>.
- [10] Dejan Glozic and Dorian Birsan. How To Keep Up To Date:
<http://www.eclipse.org/articles/Article-Update/keeping-up-to-date.html>.
- [11] Chris Grindstaff. How to use the JFace Tree Viewer:
<http://www.eclipse.org/articles/Article-TreeViewer/TreeViewerArticle.htm>.

- [12] Jakarta commons homepage: <http://jakarta.apache.org/commons/>.
- [13] Java homepage: <http://java.sun.com>.
- [14] Modest editor plugin. <http://depend.cs.uni-sb.de/index.php?446>.
- [15] Möbius homepage: <http://www.mobius.uiuc.edu/>.
- [16] MoToR homepage: <http://fmt.cs.utwente.nl/tools/motor/>.
- [17] Joe Szurszewski. We Have Lift-off: The Launching Framework in Eclipse:
<http://www.eclipse.org/articles/Article-Launch-Framework/launch.html>.