A step-by-step guide through the plugin features

Christophe Boutter

May 20, 2007

This manual 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.

This manual and some other relevant material can be found on the plugin homepage [5]. There you will find this manual, a reference to the update site needed for the installation and also the source code of the plugin.

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.

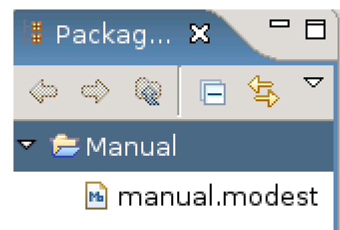
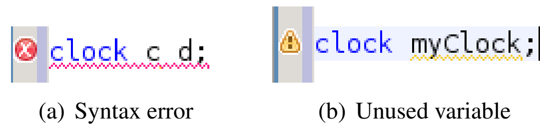


Figure 1: Explorer with MoDeST file

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).

Figure 2: Activated completion

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).



(a) Syntax error (b) Unused variable

Figure 3: Error Marker

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 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.

In the second preference page 5 the preferences for the launching of external programs (c.f. 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.

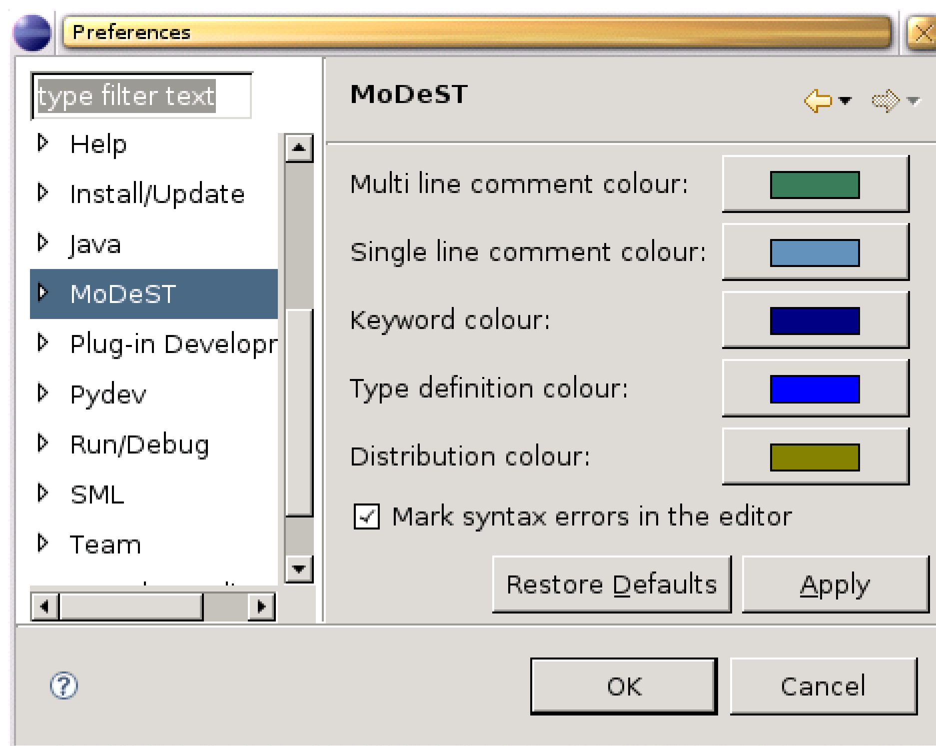


Figure 4: Main preferences for the MoDeST plugin

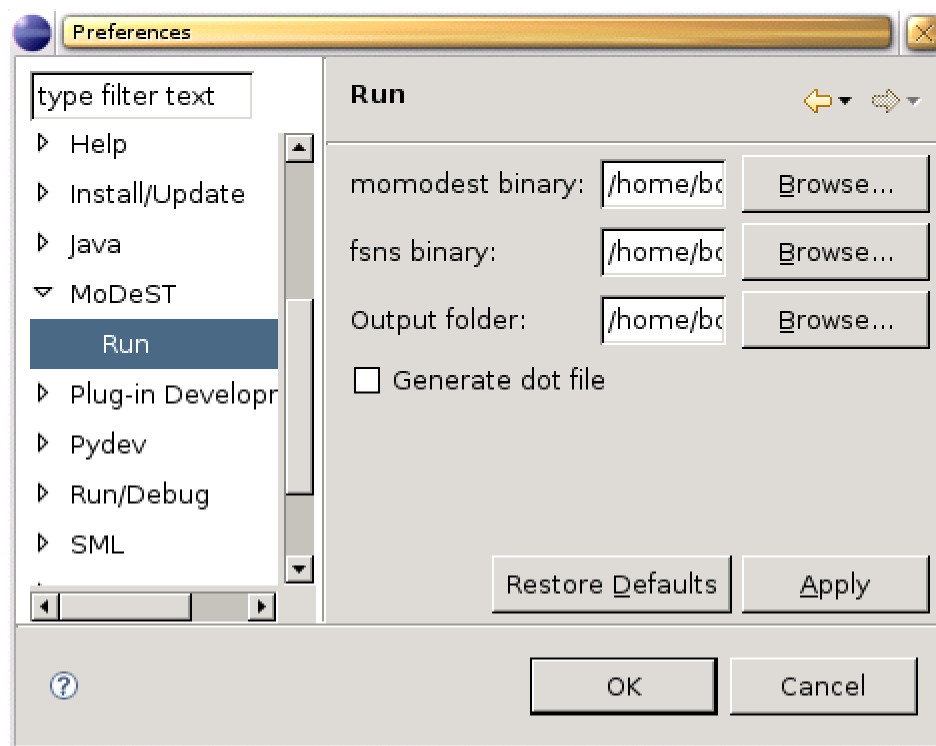


Figure 5: External program preferences

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. 5).

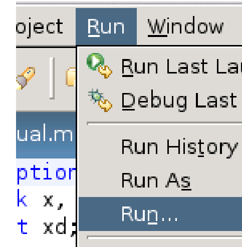


Figure 6: The run menu

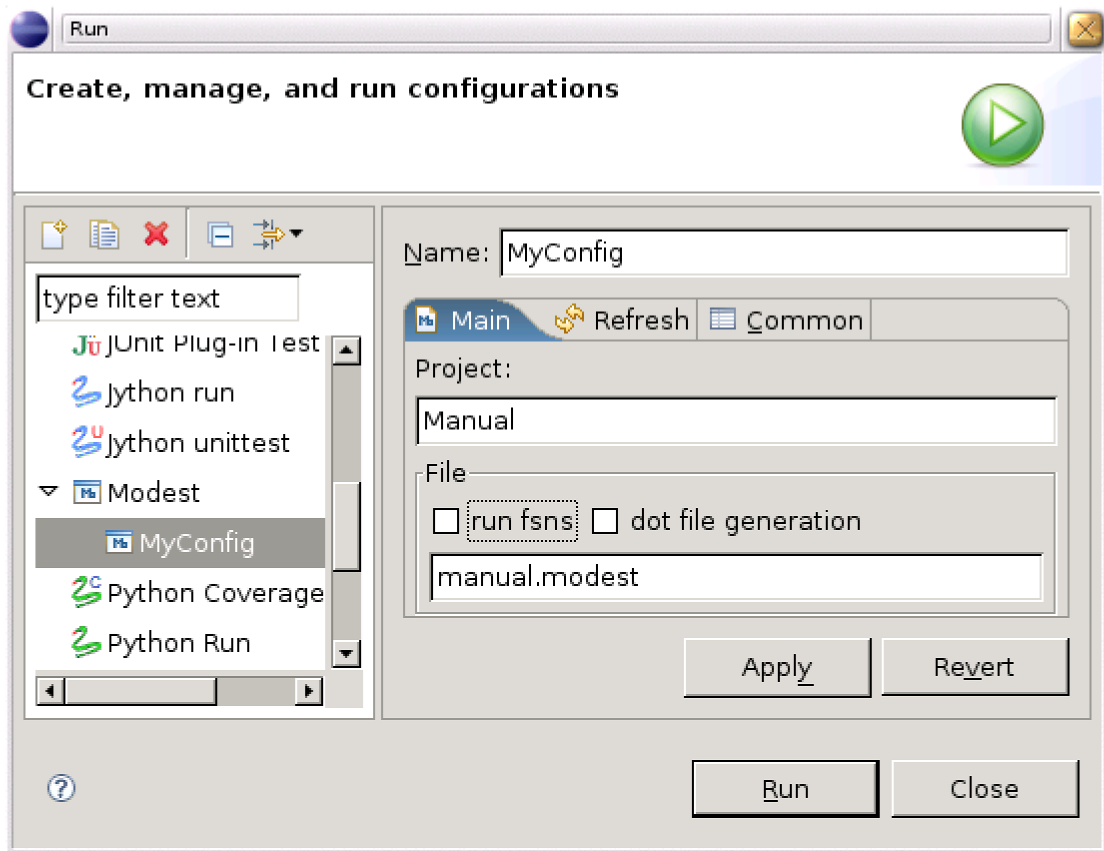


Figure 7: Launch tab

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 8.

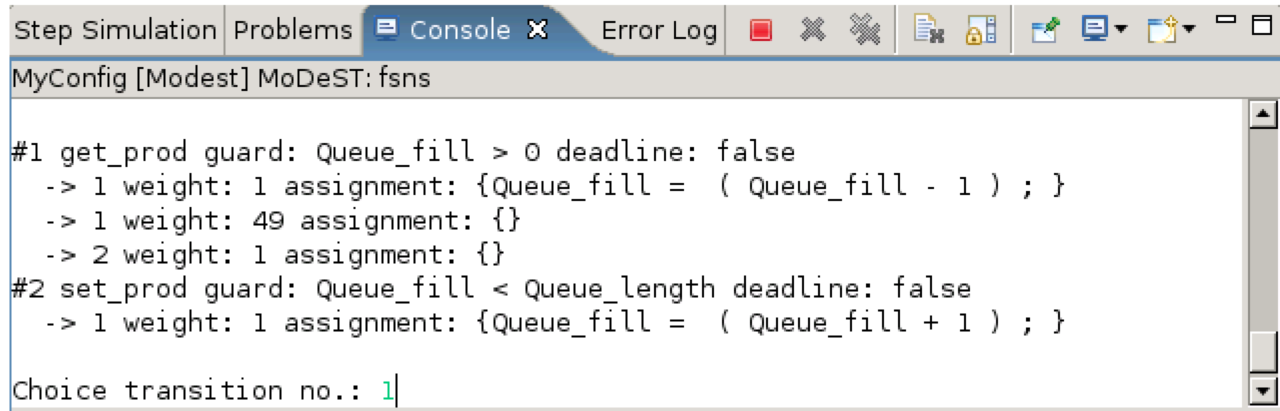


Figure 8: The FSNS interface in the console View

4 Simulating MoDeST in Eclipse

As seen in figure 8 some means to simulate MoDeST code in a stepwise fashion already exists. But the figure 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 [3]. 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.

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 9. This example is a slight adaptation of an example presented in [2].

4.1.1 Syntax introduction

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

patient action set_prod;

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



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

```
(1)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 9: The cashier code

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) firsthand. 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 `palt` construct (14), which provides a weighted *probabilistic choice*, where each *weight* has the form *:w:* (15), with *w* a positive natural number. A `palt` 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).

4.1.2 Semantics introduction

For a complete overview over the MoDeST semantics please refer to [3]. 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 [4]. 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 **??**. Such transitions are obtained with the `palt` construct which has weighted probabilistic alternatives. Such a transition is taken by first handling the action guarding the `palt` construct and then making a probabilistic choice over the alternatives of the `palt` and handling the assignment in the alternative. The whole transition is done in an atomic way and cannot be split.

In the syntax introduction 4.1.1 guards were presented. These guards are written in `when`(.) clauses and should not be confused with `if`-statements known from other programming languages. 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.

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 9 can be shown in order to demonstrate how the step simulation of the MoDeST plugin works. The code presented in the figure 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 10. This images show parts of the MoDeST Step Simulation View that is needed in order to display the transitions. This View is opened via the menu: Window -> Show View -> Other. . . -> MoDeST-> Step Simulation.

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

[fill < length] set_prod (probabilistic transition following)

[fill > 0] get_prod (probabilistic transition following)

Figure 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.

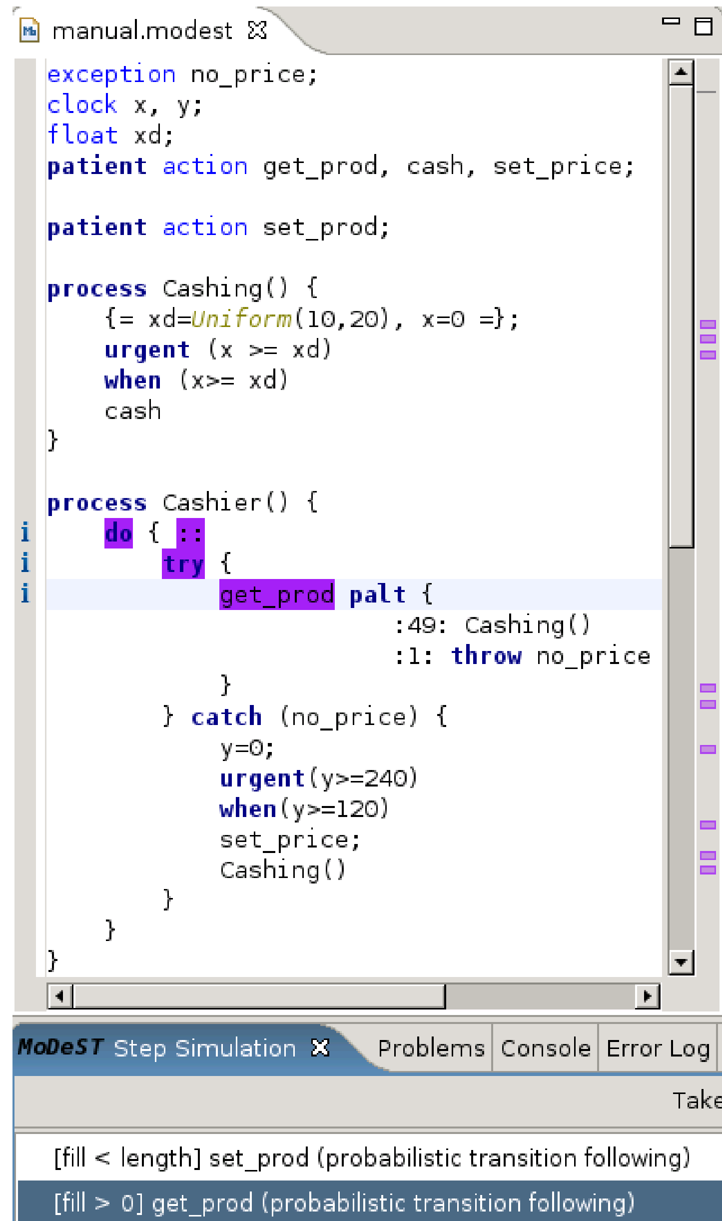


Figure 11: Since the interest is placed on the *Cashier* process the transition featuring the *get_prod* action is selected. Marker are set in the text that highlight the active parts of the code. **Important:** for the Markers to be highlighted as presented, the user needs to set the “Text as" attribute of the “Info Marker" in the “Preferences" under “General->Editors->Text Editors->Annotations" to “Highlight".

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 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.

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

[] no_price

[fill < length] set_prod (probabilistic transition following)

Figure 13: In order to show the exception handling the *throw* statement is selected since it was enabled with the probabilistic choice taken before.

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

[y >= 120] set_price

[fill < length] set_prod (probabilistic transition following)

Figure 14: The thrown exception is caught and next the first action (*set_price*) in the *catch* statement can be taken.

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

[x >= xd] cash

[fill < length] set_prod (probabilistic transition following)

Figure 15: Now the process call of the *Cashing* process presents the *cash* action in a transition.

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

[fill < length] set_prod (probabilistic transition following)

[fill > 0] get_prod (probabilistic transition following)

Figure 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.

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 [1].

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 update site [6]. 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.

References

[1] ANTLR plugin homepage: `http://antlreclipse.sourceforge.net/`.

[2] 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.

[3] Pedro R. D’Argenio, Holger Hermanns, Joost-Pieter Katoen, and Ric Klaren. MoDeST: A compositional modeling formalism for hard and softly timed systems. 2005.

[4] Java homepage: `http://java.sun.com`.

[5] Modest editor plugin. `http://depend.cs.uni-sb.de/index.php?446`.

[6] Modest editor plugin update site. `http://depend.cs.uni-sb.de/fileadmin/user_upload/eclipse`.