CoreASM Editor & Debugger — Manual

An advanced Editor and Debugger for CoreASM

http://uni-ulm.de/in/pm/projects/coreasm
https://github.com/CoreASM/
http://coreasm.org

Marcel Dausend, Markus Müller, and Michael Stegmaier
{marcel.dausend, markus.mueller, michael-1.stegmaier}@uni-ulm.de

Version 1.7.0-SNAPSHOT, March 2013

1 Introducing Notes

The CoreASM Eclipse plugin extends the Eclipse IDE for editing, debugging, and executing CoreASM specifications. This version is a major upgrade from the latest version (0.6.8.beta). It offers a reimplemented and enhanced editor which integrates the latest jparsec parser[1]. This new editor performs noticeably better than the old one and introduces some valuable features to revise specifications like quick fixes and syntax checks. Furthermore, CoreASM specifications can be investigated in an intuitive as well as comprehensive manner with the new debugger which makes use of the regular Eclipse debugging components.

The reimplementation and enhancement of the editor component has been implemented by Markus Müller during his diploma thesis [3]. The debugger has been implemented as part of a bachelor thesis by Michael Stegmaier [5] and has been introduced on the ABZ-conference 2012 in Pisa [2]. Both theses have been supervised by Prof. Dr. Helmuth A. Partsch, head of the institute of Software Engineering and Compiler Construction at the University of Ulm. The work has been initiated and mentored by Marcel Dausend. Meanwhile, the project has been merged with the official CoreASM development project (`www.coreasm.org`) and is provided as open source via github [1].

[1] `http://jparsec.codehaus.org/`

Contents

2 The CoreASM Eclipse Plugin

The daily version of the CoreASM Eclipse plugin can be received via github [1]. A guide for building and executing the development version can be found on the referred website, too. Non-developers can easily try out the latest release version of the CoreASM Eclipse plugin by themselves. It is distributed via the Eclipse Marketplace and our Eclipse Update Site. Further information can be found in our wiki at github. You are welcome to contact the authors in case of questions or for providing your feedback[2].

2.1 System Requirements

The following infrastructure is required for the CoreASM Eclipse plugin:

- Java SE Runtime Environment 7
  `http://www.oracle.com/technetwork/java/javase/downloads/index.html`
- Eclipse IDE for Java Developers (version *Kepler* suggested)
  `http://www.eclipse.org/downloads/`

This version of the CoreASM Eclipse Plugin has been developed and tested under

Ubuntu Linux 64bit v14.10 &
Windows 7 and 8.1
with
Kepler Service Release 2 64 bit &
Luna Eclipse Standard 4.4 64 bit
Oracle Java SE JDK 7

2.2 Installing CoreASM Eclipse Plugin

The Plugin can be installed either from the Eclipse Marketplace or by performing the following steps:

- Check if the required software (see above) is already installed on the target machine and if not, install the software.
- Open the Help-menu inside Eclipse
- Select the menu item Help » Install New Software...
- Paste the url of this site `http://webcoreasm.informatik.uni-ulm.de/coreasm-repository` into the field "work with" and press ENTER
- Next press Select All- and afterwards Next-button

[2] please send a request by e-mail to `marcel.dausend@uni-ulm.de`

- Confirm the selection of the "CoreASM Eclipse Plugin" for installation by pressing the Next-button
- Accept the license and start the installation by pressing the Finish-button
- When the warning appears that you are installing unsigned content, you have to press the Okay-button to continue
- Last, you have to restart Eclipse so that the "CoreASM Eclipse Plugin" becomes available to you

If you like, you can build CoreASM by your own using the sources on github. The sources and our wiki are available at `https://github.com/CoreASM/coreasm.core`.

3 General Introduction to CoreASM and its Editor

The CoreASM plugin for Eclipse offers two components which are designed to support writing CoreASM specifications: The redesigned editor (see fig. 1 – middle) and an outline view of the currently open CoreASM specification (see fig. 1 – right). Moreover, a view showing the AST of the current specification is provided to assist, for instance, in plug-in development.

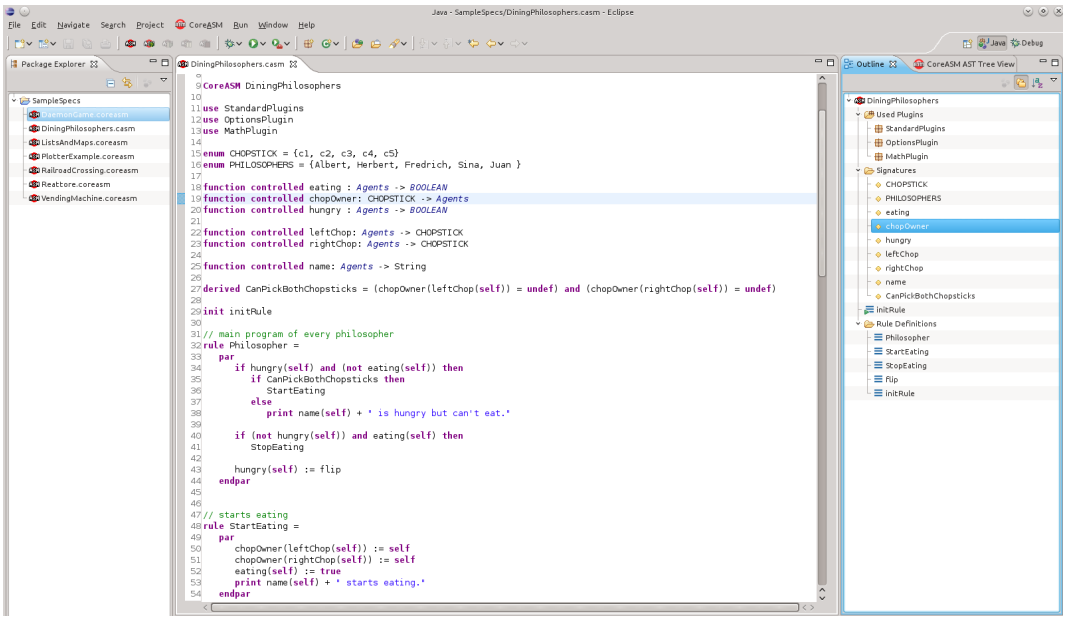


Figure 1: Overview of the Eclipse IDE, showing the CoreASM editor and its outline view

The editor offers a lot of features to support the user to create, examine, and correct or revise specifications:

- syntax highlighting
- syntax checking
- warning and error markers (which are also shown in the eclipse problems view)
- quick fixes for several issues
- tooltips showing parser information
- bracket highlighting
- code completion
- ...

The outline view shows an overview of the specification corresponding to the currently open editor. The user can decide if the entries should be shown in a structured way, where use-statements, signatures and rules are grouped or in a flat representation. Also, the user can decide if the entries should be ordered alphabetically or in textual order of the specification. The buttons at the top of the outline view can be used to toggle between those configurations. Moreover, entries in the outline view can be used to navigate to their corresponding definition inside the specification by simply clicking on the desired entry. If the current specification cannot be parsed correctly, the outline view is marked as outdated. In this case, the user is advised to correct the specification before he can continue to use the outline view.

3.1 Creating a Specification

To create a CoreASM specification, an existing project in the current eclipse workspace is required as a container for the new specification. A new project can be created in three steps:

1. Choose File » New » Project... from the Eclipse menu or press Ctrl + N.
2. Choose General » Project from the *New Project* dialog.
3. Give the new project a name and press Enter or click on the *Finish*-button.

A CoreASM specification can be created in at least two ways: One way is creating a new text file with a name ending on `.casm` or `.coreasm` within the project of choice. An alternative is the *new*-wizard:

1. Choose File » New » Others... from the Eclipse menu or press Ctrl + N.

2. In the appearing *New*-dialog select CoreASM ⟩ CoreASM Specification and press Enter or click on the *Next*-button.

3. Preferably select a project from the workspace as a container for the specification. This can be done by either using the file selection dialog or manually entering the project's name.

4. Give the new specification file a name ending with `.casm` or `.coreasm` and press Enter or click on the *Finish*-button.

The structure of a CoreASM specification and the CoreASM language are described in "CoreASM Language User Manual" [4, p. 7][3]. A "Hello World!"-example is given in section 7.

3.2 Executing a Specification

A CoreASM specification can be executed using Carma [4, p. 4][4] or the CoreASM Eclipse plugin. There are two ways to execute a CoreASM specification in Eclipse:

The easiest way to run the specification of the currently selected editor component is to click on the green run-button of the eclipse toolbar (see fig. 2(a) and fig. 2(b)). As a result, the selected specification is executed by the CoreASM engine and the output is shown inside the *Console*-view of Eclipse (see fig. 2(e)). Running a specification the first time automatically creates a *Run Configuration* which specifies some options for the execution of the related CoreASM specification. Fig. 3 on page 8 shows the default *Run Configuration* for the DiningPhilosophers specification. The different options for a certain specification configure the termination condition for a CoreASM execution and the verbosity of its output.

The second option to start a specification is to use a *Run Configuration*. If a *Run Configuration* for a specification exists, or after it has been created, its specification can be executed by selecting it. The down-arrow on the right-hand side of the *Run*-button or the *Run Configuration*-menu can be used to open a selection list (see fig. 2(c)). To access a *Run Configuration* via the Eclipse menu, select Run ⟩ Run Configurations...

The execution of a specification can be paused, resumed, and stopped by clicking one of the buttons (see fig. 2(a)) located under the CoreASM-menu or selecting an entry from that menu (see fig. 2(d)).

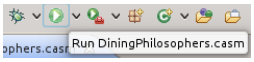
4 Debugging a Specification

A CoreASM specification can be started for debugging by either using the de*bug*-button of the toolbar (see fig. 2(a)) or by selecting a *Debug Configuration* via the down-arrow beside this button. Another option is using the eclipse menu Run ⟩ Debug Configurations...

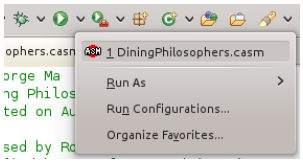
[3] `http://coreasm.svn.sourceforge.net/viewvc/coreasm/engine-carma/trunk/doc/user_manual/CoreASM-UserManual.pdf`

[4] Carma can be received at `www.coreasm.org/download`

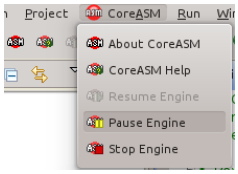
(a) Relevant excerpt of the Eclipse tool bar



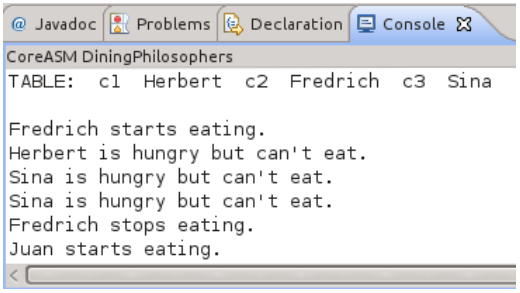
(b) Running the DiningPhilosophers specification



(c) ... by selecting its *Run Configuration*



(d) The CoreASM menu



(e) The output of the DiningPhilosophers specification

Figure 2: Running a CoreASM specification

If the specification has been paused by using one ot the pause-buttons, or a break point has been reached, Eclipse asks the user to switch to the debug perspective. Confirming this question, the user will be shown a screen similar to fig. 4. This debug perspective contains the views (from top left to bottom right) described in table 4 on page 10.

The debbuging of CoreASM specifications is described in detail in the following sections.

4.1 Stepping Through a Specification

Once the execution of a specification in debug mode is paused, one can analyze a specification step-by-step. There are three different kinds of stepping, which can be forced by pressing the related buttons in the toolbar of the debug perspective or use a keyboard shortcut:

- — F7 Step Return
  Executing all statements and stopping at the next sequential block
- — F6 Step Over
  Executing a single step of the machine
- — F5 Step Into
  Executing a single step, which can also be a step inside a sequential block

Debugging of imperative languages differs in many points from debugging Abstract State Machines. One difference is, that in a CoreASM specification without sequential parts, all

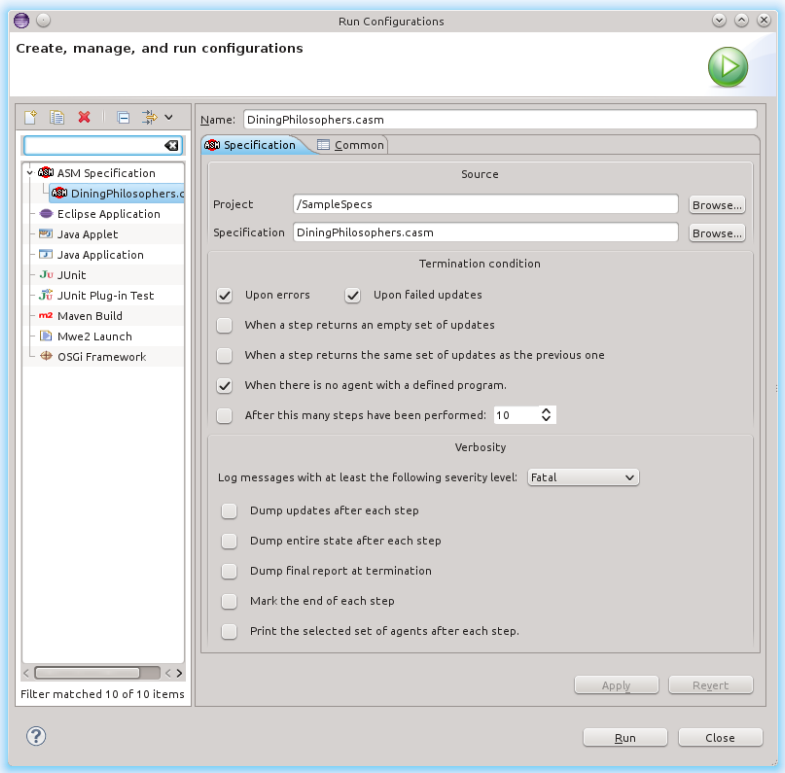


Figure 3: Default *Run Configuration* for the DiningPhilosophers specification

different step actions result in collecting and aggregating all updates of the current step. The execution will stop again before the first statement of the next step will be computed, so that one can examine the update set before the state of the machine is updated. To continue the execution of the interpreter click on one of the resume-buttons .

4.2 Adding/Removing Breakpoints

A breakpoint can be set from within the source editor by double-clicking on the ruler or right-clicking it and selecting Toggle breakpoint (see Fig. 5).

There are three different types of breakpoints (see Fig. 6):

- Watchpoints: They will be added if the selected line starts with “function” or “universe”. They will suspend the execution whenever the value of a function/universe changes or is being read.

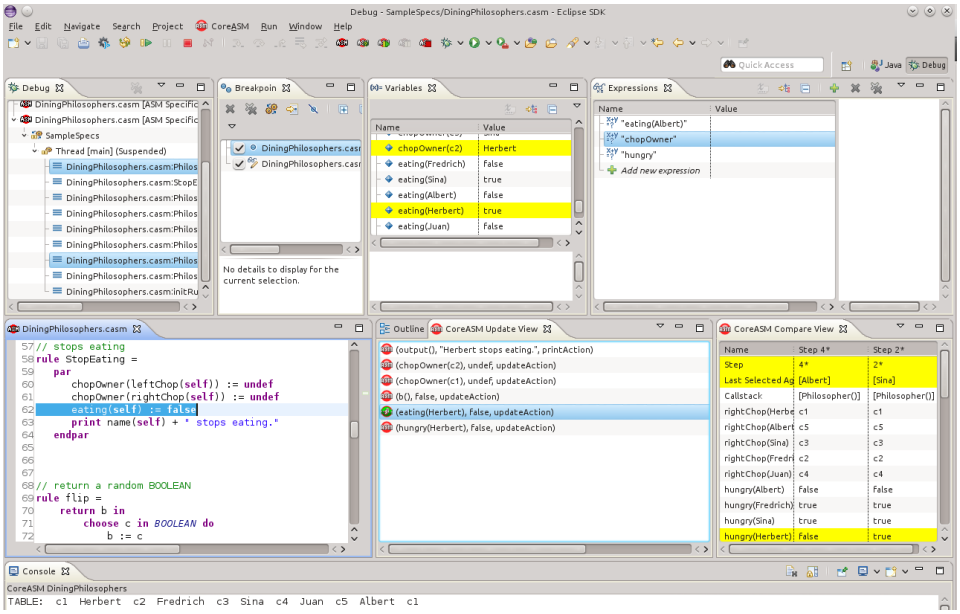


Figure 4: Overview of the Debugger

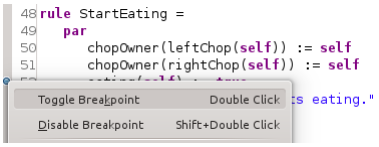


Figure 5: Line breakpoint set at line 52 of the DiningPhilosophers specification.

- Method breakpoints: They will be added if the selected line starts with `rule`. They will suspend whenever an update occurs from a line within the selected rule's body.
- Line breakpoints: They will be added if none of the above breakpoints can be added. They will suspend whenever an update occurs from the selected line.

A breakpoint can be disabled by choosing Toggle Breakpoint from context menu of the ruler or by un-checking the box in front of its entry in the Breakpoints view. By enabling the *Skip All Breakpoints*-toggle switch, all breakpoints are discounted during an execution without the need of changing the set of active breakpoints.

4.3 Watching Functions and Expressions

The *Variables*-view (see Fig. 7(a)) allows to watch and examine the values of all available functions of the machine's state. All values that have been changed due to the last update-set

Table 1: Overview of the debugging components in fig. 4

View	**Description**	**Interaction**
Debug	Shows the currently executed specification and its steps	Selected steps are taken into account for the CoreASM Compare View
Breakpoints	Lists all Breakpoints of the workspace	Breakpoint(s) can be disabled or re-enabled, skipped, deleted, exported, imported and used to navigate to its related source destination
Variables	Shows the state of the CoreASM engine	The state of the CoreASM execution can be investigated and manipulated
Expressions	Shows user defined CoreASM expressions and their values	User defined expressions are passed to the interpreter and evaluated based on the current state of the execution
Editor	Shows the statement to be evaluated next	Changes to the specification during debugging do not influence the current execution
Update View	Shows all updates, optionally restricted to a specific agent, which are collected up to now during the current step of the interpreter	An update can be used to navigate to the statement of its origin; Updates which correspond to a breakpoint are highlighted by a green symbol .
Compare View	Shows the state of the CoreASM execution for specific steps	All selected steps in the *Debug*-view are shown for comparison; Optionally, just differences are presented.

are highlighted in yellow color. The value of a function at a specific location can be changed by clicking on its value entry, changing the value by modifying the text, and pressing Enter. This modification is applied directly to the state of the machine and will not induce an extra update — this feature has to be used with caution.

To keep an eye on a specific function for a given location, a corresponding entry in the *Expressions*-view (see fig. 8(a)) can be created by right-clicking on the desired entry and selecting Watch from the menu (see Fig. 7(b)).

Additional expressions can be added to the *Expressions*-view by pressing Add new expression and entering either a universe name, or function name and its location (see fig. 8(b)). Entering a function name without its location (e.g. hungry) will result in showing a container of all it's locations and values (see Fig. 8(a)). Expressions can be removed by selecting at least one entry and pressing Del., or using the buttons .

Another way to inspect expressions on the fly is marking an expression inside the *Editor*-view or moving the mouse over a single statement. By doing so, a tooltip will be presented that shows the result of the evaluation of the marked expression based on the current context of the machine's evaluation, i. e. the global state and the current computation context. Two examples are given in fig. 9.

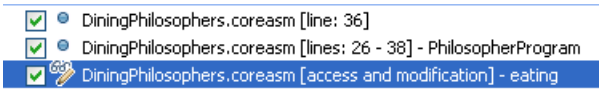
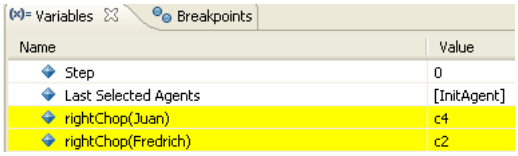
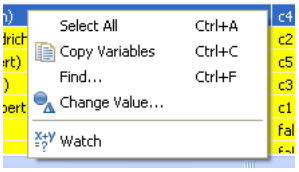


Figure 6: Three different kinds of breakpoints listed in the Breakpoints view.



(a) The *Variables*-view shows all functions at its location, their value, definition type, and current type.



(b) Context-menu of the *Variables*-view.

Figure 7: The *Variables*-view for inspecting function and modifying their values.

5 Taking Care of Updates

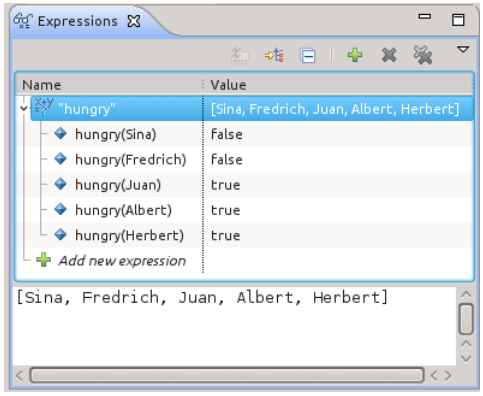
The *Update*-view (see fig. 10) lists all updates which have been computed during the current step. Each update is a triple, consisting of the function and its location, the value, and the type of action. The action type is an internal CoreASM specific value. By double-clicking on an update-entry, the source of this update inside the specification is presented to the user. Updates that correspond to an active breakpoint are marked by a green symbol . To focus on the updates of a specific agent, a filter option can be applied (see fig. 11(a)).

6 What has been changed?

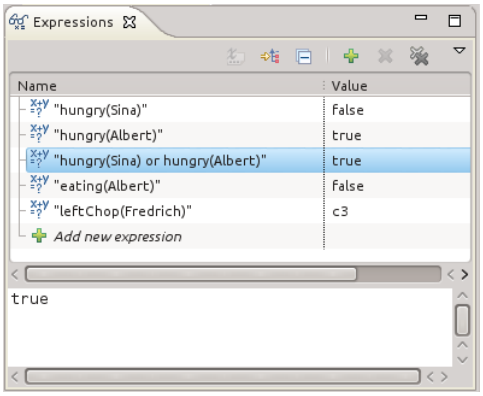
The *Compare*-view enables analyzing the state of the machine over the time. Therefore, the view shows all functions and their values for selected steps of the machine side-by-side. The selection has to be performed within the *Breakpoints*-view where all steps are listed. Multiple steps can be selected while holding Ctrl (for single selection) or holding ⇧ (to mark a range of steps). Steps, which are marked by a * are intermediate steps resulting from sequential steps. To clear up the *Compare*-view the filter option can be used, which results in hiding all corresponding functions with equal values (see fig. 11(b), p. 14).

7 Excursus — Modules in CoreASM: Hello World

The following specification `HelloWorld.casm` implements an extended "Hello World!". This specification itself specifies the output of "we proudly present:". It further demonstrates the use of modules by including the module `PrintHelloWorld.casm` which implements the output of "Hello World!". As a result, the rule `PrintHelloWorld` can be called from

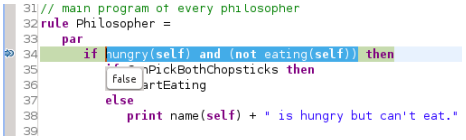


(a) A function and its values for each location.

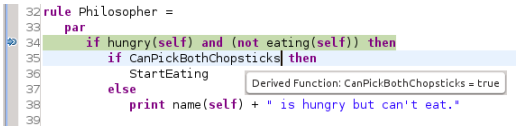


(b) Some functions of specific locations and their values. The marked entry shows a user defined expression and its value.

Figure 8: The *Expressions*-view shows user selected universes, functions for either all or one specific location, and evaluates user defined expressions depending on the current state.



(a) The expression at the line breakpoint is evaluated on-the-fly.



(b) The result of the derived function `CanPickBothChopsticks` is presented as a tooltip when the mouse cursor hovers over its calling statement.

Figure 9: During debugging, expressions can be marked inside the *Editor*-view so that the result will be shown as a tooltip.

the initial rule of the main specification “HelloWorld”.

```
1  /** A multi-line comment
2  *   for the HelloWorld specification
3  *   Each specification has to start with CoreASM <name>
4  */
5  CoreASM HelloWorld
6
7  //A single line comment previous to the block of used plugins
8  use Standard
9  use Modularity
10
11 //the initial rule definition
```

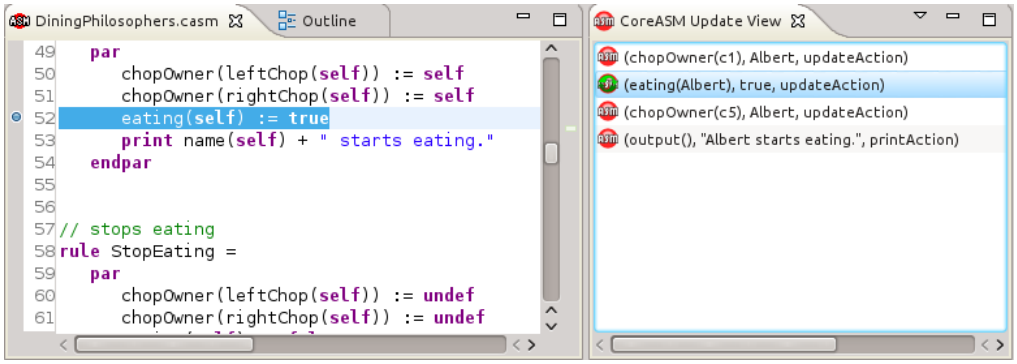


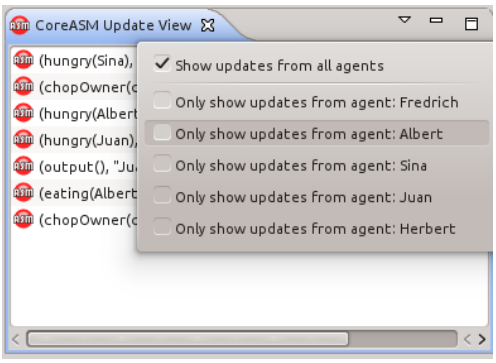
Figure 10: The *Editor* (left) and *Update*-view (right): The statement in the specification causing the marked update is highlighted. It is located in a line with an active breakpoint.

```
12 init HelloWorld
13
14 /** The path to an included CoreASM module
15 *   has to be given within double quotes.
16 */
17 include "./PrintHelloWorld.casm"
18
19 rule HelloWorld =
20 seq
21         print "we proudly present:"
22 next
23         PrintHelloWorld()
```

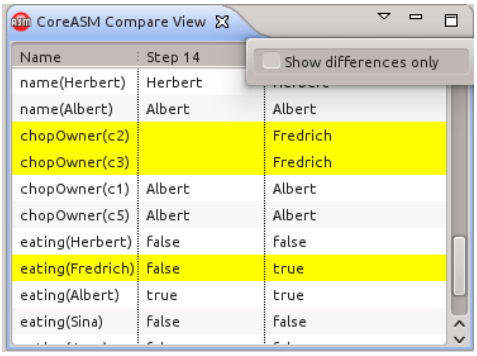
The module `PrintHelloWorld.casm` implements the output of "Hello World!". In difference to a main specification, its header starts with `CoreModule <module name>` and an init rule is not allowed here.

```
1 //modules can be included in CoreASM specifications
2 CoreModule PrintHelloWorld
3
4 use Standard
5
6 rule PrintHelloWorld =
7         print "Hello World!"
```

Further example specifications, e. g. the DiningPhilosophers specification, are part of the distributable and can be found in the folder `sampleSpecs`.



(a) The filter of the *Update*-view helps the user to concentrate on the updates of a specific agent.



(b) The *Compare*-view shows the functions for different states side-by-side. To focus on the changes between those states, the filter can be used to hide functions with equal values.

Figure 11: Both, the *Update*- and the *Compare*-view offer filters to focus on certain aspects.

References

[1] M. Dausend, R. Farahbod, M. Müller, A. Raschke, and M. Stegmaier. The CoreASM Project. https://github.com/CoreASM/coreasm.core, 2012.

[2] M. Dausend, M. Stegmaier, and A. Raschke. Debugging Abstract State Machine Specifications: An Extension of CoreASM. In *Proceedings of the Posters & Tool demos Session, iFM 2012 & ABZ 2012*, 2012.

[3] M. Müller. Entwicklung eines Editors für CoreASM — Redesign nach Software Engineering Methoden. Diplomathesis, University of Ulm, 2012.

[4] R. Rarahbod. *CoreASM Language User Manual*, engine version 1.5-beta edition, 2011.

[5] M. Stegmaier. Entwurf und Implementierung eines Debuggers für Abstract State Machines in CoreASM. Bachelorthesis, University of Ulm, 2012.