CoreASM Language User Manual

engine version 1.6.5-beta

github.com/CoreASM/

Roozbeh Farahbod info@coreasm.org

Marcel Dausend
marcel.dausend@uni-ulm.de

Copyright © 2006-2014

DRAFT of March 17, 2016

This document is still under construction to match the latest version of the engine. Your criticism is welcome!

Contents

1	Inst	talling CoreASM	4
	1.1	CoreASM with Carma	4
		1.1.1 System Requirements	4
		1.1.2 Installing Carma	4
		1.1.3 Running Carma	4
	1.2	CoreASM Eclipse Plugin	4
		1.2.1 System Requirements	4
		1.2.2 Installing CoreASM Eclipse Plugin	5
		1.2.3 Using CoreASM Eclipse Plugin	5
	1.3	Using CoreASM Compiler	7
2	Core	eASM Specification	7
	2.1	Running CoreASM Specifications	9
3	Ker	rnel	10
	3.1	Rule Forms	11
	3.2	Kernel Engine Properties	11
4	Bas	sic ASM Plugins	12
	4.1	Block Rule	12
	4.2	Choose Rule	13
	4.3	Conditional Rule	13
	4.4	Forall Rule	14
	4.5	Let Rule	14
	4.6	Case Rule	15
	4.7	Predicate Logic	15
	4.8	Number Background	16
5	Sta	ndard Plugins	18
	5.1	Kernel Extensions	18
	5.2	Abstraction	19
	5.2	Extend Pula	10

CONTENTS

	5.4	TurboASM Rules	20
	5.5	String Background	22
	5.6	Input and Output	23
	5.7	Collection	24
	5.8	Set Background	25
	5.9	Bag Background	27
	5.10	List Background	28
	5.11	Queue	31
	5.12	Stack	31
	5.13	Map Background	32
	5.14	Signature Plugin	33
6	Add	litional Plugins	34
	6.1	Modularity	35
	6.2		
	0.2	Options	35
	6.3	Options	35 35
	• • •		
	6.3	Scheduling Policies	35
	6.3 6.4	Scheduling Policies	35 36
	6.3 6.4 6.5	Scheduling Policies	35 36 37
	6.3 6.4 6.5	Scheduling Policies	35 36 37 38
	6.3 6.4 6.5	Scheduling Policies Time	35 36 37 38 38
	6.3 6.4 6.5	Scheduling Policies Time DebugInfo Math 6.6.1 Constants 6.6.2 Basic Derived Functions	35 36 37 38 38

1 Installing CoreASM

There are currently two user interfaces available for the CoreASM engine: a command-line interface called Carma, and a graphical interactive development environment in the Eclipse platform, known as the CoreASM Eclipse Plugin.

1.1 CoreASM with Carma

All sources of the CoreASM engine with Carma can be downloaded at https://github.com/CoreASM/coreasm.core.

1.1.1 System Requirements

You need to have Sun Microsystems Java 1.6 (JVM) installed on your machine.

1.1.2 Installing Carma

You can build CoreASM with Carma using the maven build system provided in the source package.

1.1.3 Running Carma

Under Carma's home directory (where you installed Carma), simply run 'carma' (under POSIX systems) or 'carma.bat' (under Windows systems). To be able to run Carma form other directories, change the value of CARMA_HOME environment variable in 'carma' or 'carma.bat' (depending on your operating system) so that it points to the folder in which Carma is installed.

To start, try Carma with '-help' to see the list of command-line arguments.

1.2 CoreASM Eclipse Plugin

This section explains how to install the CoreASM Eclipse plugin.

1.2.1 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

1.2.2 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
- 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.

1.2.3 Using CoreASM Eclipse Plugin

Creating a New Project

- 1. From the Eclipse menu choose: File New Project...
- 2. Choose General Project from the "New Project" dialog. Click Next.
- 3. Give the project a name. Click Finish.

Creating a New CoreASM Specification

Method 1:

- 1. From the Eclipse menu choose: File New Other...
- 2. In the New dialog choose CoreASM Specification. Click Next
- 3. Choose the project container for the specification.
- 4. Enter the name of the new CoreASM specification file. The file must have the extension .casm or .coreasm.
- 5. Click Finish

Method 2:

- 1. File the Eclipse menu choose: File New File...
- 2. In the new file dialog choose a project container for the new file and enter a name for the new file. Again, The file must have the extension .casm or .coreasm.
- 3. Click Finish

Running a CoreASM Specification

Method 1:

Shortcut method for running a specification with default configuration:

- 1. Press the "play"-button that is usually used to compile and run programs, e.g. in Java or
- 1. In the Eclipse window, right click on a CoreASM specification file.
- 2. In the context menu choose: Run as... CoreASM Specification

Method 2:

If you need more control of the parameters for repeated execution, you can create a specific CoreASM Launch Configuration as follows:

- 1. From the Eclipse menu choose: Run Run...
- 2. In the "Run" dialog, choose the "ASM Specification" launch configuration group and create a new ASM launch configuration (right click then select New), or click the New launch configuration button on the tool bar).
- 3. Enter a name for the launch configuration.

- 4. Enter the project and specification file to be run. This can be done via the browse buttons.
- 5. Configure the "Termination Conditions" and "Output Verbosity" options as desired.
- 6. Click Apply
- 7. Click [Run] to run the specification.

Once the configuration has been launched once, it can be run again through the Run Button/Drop down menu in the main Eclipse toolbar.

Controlling the Execution of the CoreASM Engine

While the engine is running, you can click on the "Stop CoreASM Engine" button to stop the run. To pause a running engine, click on the "Pause CoreASM Engine" button. If you pause the engine, the run can be resumed by clicking on the "Resume CoreASM Engine" button.

For more control, you are welcome to use the CoreASM debugger. Further details about debugging CoreASM specifications are described in the manual of the CoreASM debugger.

1.3 Using CoreASM Compiler

The CoreASM Eclipse plugin contains a compiler, which compiles a specification into an executable jar archive. Only a subset of the CoreASM plugins described in this manual is currently compilable, but all standard plugins can be used (some with restrictions, see section 7 for more information). It is recommended to verify specifications using the CoreASM Engine, as the compiler does not provide further debugging features.

Launching the compiler The compiler can be launched by right-clicking on a specification, selecting <code>Export</code> and then clicking on <code>CoreASM</code> to <code>Jar</code> <code>Export</code> in the <code>CoreASM</code> section. This will open the configuration dialog for customization of the compilation process. Pressing the <code>Finish</code> Button will start the compiler. Any generated warnings and errors will be displayed after the process has finished. If the operation was successful, the compiler will have generated an executable jar at the configured location.

Configuring the compiler The compiler can be configured to include different logging messages and termination conditions. Further options change the paths used for the output and preprocessor manipulation. Table 1 lists all options found in the configuration dialog.

2 CoreASM Specification

Figure 1 shows a typical structure of a CoreASM specification ¹. Every specification starts with the keyword CoreASM followed by the name of the specification. Plugins that are required in the

¹ As of version 1.1, this structure is not required anymore and different components of the specification can appear in any order. The only requirement is that the specification must start with a CoreASM phrase.

option	description
Specification Name	The path to the specification. Should not
	be changed and will be filled in automatically
outputFile	The file name for the generated jar
keepTempFiles	Whether the compiler should keep generated java
	sources (location will be displayed at the end
	of the compilation)
remove Existing Files	Whether files already existing in the
	temporary directory should be removed
terminateOnError	Whether the program should terminate
	on errors. Currently always true
terminateOnFailedUpdates	Whether the program should terminate
	on failed updates
terminateOnEmptyUpdate	Whether the program should terminate
	upon generating an empty update in a step
terminate On Same Update	Whether the program should terminate
	upon generating the same updates in two steps
terminateOnUndefAgent	Whether the program should terminate
	when there is no agent with a runnable program
terminateOnStepCount	Whether the program should terminate
	after a certain number of steps
logUpdatesAfterStep	Whether the generated updates should
	be logged after each step
logStateAfterStep	Whether the complete state should be
	logged after each step
logEndOfStep	Whether the end of a step should be
	logged
logAgentSetAfterStep	Whether the selected agent set should
	be logged after a step
noCompile	Whether the compiler should generate
	a jar archive or nor
logTimings	Whether the compiler should display
	timing information
preprocessorRuns	How many times the preprocessor is
	allowed to run before generating an error
${\bf hide Core ASMOutput}$	If the compiler should hide messages
	generated by the CoreASM Parser

Table 1: Compiler options

specification are then listed one by one with the keyword use followed by the name of the plugin.

The *Header* block is where various definitions take place. What goes into this section depends on the plugins that are used. The CoreASM Kernel does not define anything for the header section.

The *init rule* of the specification (the rule that creates the initial state) is defined by keyword **init** followed by a rule name. This would be the rule that initializes the state of the machine that is defined by the specification. The body of the init rule must be declared in the *Rule Declaration* block.

A sample CoreASM specification is presented in CoreASM-Says-Hello example.

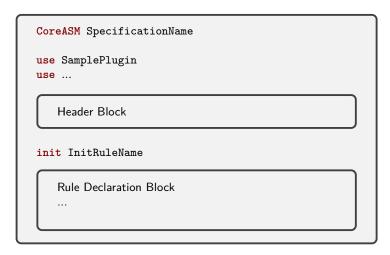


Figure 1: Typical Structure of a CoreASM Specification

2.1 Running CoreASM Specifications

To run a CoreASM specification you need to have a CoreASM engine driver. Currently, there are two engine drivers available:

- CoreASM Eclipse Plugin is a plugin for the Eclipse (see www.eclipse.org) development environment that provides syntax highlighting and a nice GUI to control specification runs.
- Carma is a command-line CoreASM engine driver. To run a specification using Carma simply run Carma on the command line and pass it the name of the specification file as an argument. Make sure to specify a termination condition (e.g., --steps 20 or --empty-updates) for the run. Run Carma with --help for a complete list of options that controls its behavior.

The following command runs MySpec using Carma and stops after 30 steps, or after a step that generates empty updates; it also dumps the final state before termination.

```
carma --steps 30 --empty-updates --dump-final-state MySpec.coreasm
```

Alternatively, to run the specification of CoreASM-Says-Hello example, one can use the following options which would make Carma to mark the end of each step and stop after 30 steps or when there is no agent with a defined program:

```
carma --marksteps --steps 30 --no-agent ThisIsCoreASM.coreasm
```

In this example, Carma will stop after three steps.

```
CoreASM-Says-Hello example
   CoreASM ThisIsCoreASM
   use Standard
   init InitRule
   rule InitRule =
      par
          terminate := false
          program(self) := @MainProgram
      endpar
   rule MainProgram =
      if not terminate then
             print "This is CoreASM."
             terminate := true
         endpar
      else
          program(self) := undef
```

3 Kernel

Kernel of the CoreASM engine provides the minimum set of vocabulary and rules to have a CoreASM specification.

Basic values such as undef, true, and false are defined in the kernel along with the background of Boolean values (BOOLEAN) and the universe of AGENTS. A function called program is also defined in the kernel which maps agents to their programs (CoreASM rules). At any time during the evaluation of a rule, self refers the the agent that is running the enclosing rule.

CoreASM kernel also defines a couple of important operators:

This is the equality operator.

▷ ruleelement id Kernel

This operator returns the rule element of a rule with the given name (id). Rule element is an element in the CoreASM state that represents a rule defined in the specification. It is useful in assigning rules to programs of agents. In the following example, Main is the name of a rule:

```
program(self) := ruleelement Main
```

The above rule, assigns the rule named Main as the value of the program of the agent running this rule.

ho 0 id Kernel

Returns the rule element (rule body) or function element of a rule or function with the given name (*id*). If the given name is the name of a rule, it works exactly the same as **ruleelement**. Thus, if Main is rule, we can have:

```
program(self) := @Main
```

3.1 Rule Forms

The following rule forms are defined in the kernel:

Assigns the value of value to the location loc.

▶ import id do rule Kernel

Imports a new element, assigns it as the value of the environment variable id, and evaluates rule.

► skip Kernel

Does nothing. This is like a NoOp.

3.2 Kernel Engine Properties

The following properties affect the behavior of the CoreASM engine.

- engine.error.printStackTrace if equals to "yes", the engine will print the stack trace of errors
 and exceptions. The default value is "no".
- **engine.limits.maxProcessors** the maximum number of processors the engine can use for simulation. The default value is "1".
- scheduler.printProcessorStats if equals to "yes", the engine will print some information on processor utilization after every step. The default value is "no".
- scheduler.threadBatchSize in a multi-threaded simulation, the value of this property defines the minimum number of agents assigned to every thread. The default value is "1".
- engine.pluginFolders a colon-separated list of folders that provide additional plugins.
- engine.pluginLoadRequest a comma separated list of plugins to be loaded in addition to those listed in the specification being loaded.

4 Basic ASM Plugins

In this section we list the plugins that provide the basic ASM rule forms. All the plugins in this section can be loaded individually (as instructed in each section) or all together with the following use phrase,

```
use BasicASMPlugins
```

which automatically loads the following plugins: BlockRule, ConditionalRule, ChooseRule, Forall-Rule, LetRule, and Number.

Note that the words "Plugin" and "Plugins" in the name of the plugins are optional. For example, Basic ASM plugins can also be loaded using the following line:

```
use BasicASM
```

4.1 Block Rule

The Block Rule plugin can be loaded by the following use phrase:

```
use BlockRule
```

This plugin provides the following rule form:

```
ightharpoonup par rule_1 \underline{rule_2 \dots rule_n} endpar Block Rule Plugin \underbrace{optional}
```

Instructs the engine to evaluate all the given rules in parallel. The update generated by this rule is the union of all the updates generated by $rule_1$ to $rule_n$.

4.2 Choose Rule

The Choose Rule plugin can be loaded by the following use phrase:

```
use ChooseRule
```

This plugin provides the following rule form:

Chooses an element from the *enumerable*² value which satisfies guard, assigns it as the value of id and evaluates rule₁1. If the ifnone clause is provided, rule₂ will be evaluated if no element can be found. The last keyword endchoose is optional.

The following example chooses the minimum price p from the set of prices and prints (see Section 5.6) p on the screen:

```
choose p in prices with (forall pi in prices holds p <= pi) do
print p</pre>
```

See Section 4.7 for more information on forall expressions.

The ChooseRule plugin also provides the following expression form to non-deterministically pick a value from an enumerable that satisfies the given (optional) condition:

```
    ▷ pick id in value with guard optional
    ○ pick id in value with guard Optional
```

For example, the following assignment non-deterministically assigns **true** or **false** to location **foo**:

```
foo := pick x in {true, false}
```

4.3 Conditional Rule

The Conditional Rule plugin can be loaded by the following use phrase:

```
use ConditionalRule
```

²An *enumerable* is an element that can be enumerated; i.e., that is a collection of other values. Sets, universes, and some backgrounds are enumerable values.

This plugin provides the following rule forms:

Evaluates *rule* only if *value* is **true**. It expects *value* to be a Boolean value (being either **true** or **false**).

Evaluates rule₁ only if value is true and rule₂ only if value is false. It expects value to be a Boolean value (being either true or false).

The Conditional Rule plugin also provides a conditional operation of the form:

The value of this operator is $value_t$, if $value_c$ evaluates to true; it is $value_f$, if $value_c$ evaluates to false; otherwise, it is undef.

4.4 Forall Rule

The Forall Rule plugin can be loaded by the following use phrase:

```
use ForallRule
```

This plugin provides the following rule form:

For all the elements in the enumerable *value* that satisfy *guard*, assigns the element to *id*, and evaluates *rule*. The following examples assigns the DefaultProgram rule as the program of all the agents program of which is <u>undef</u>:

```
forall a in AGENTS with program(a) = undef do
    program(a) := ruleelement DefaultProgram
```

4.5 Let Rule

The Let Rule plugin can be loaded by the following use phrase:

```
use LetRule
```

This plugin provides the following rule form:

For all the given pairs of id and value, assigns $value_i$ as the value of the environment variable id_i , and evaluates rule.

4.6 Case Rule

The Case Rule plugin can be loaded by the following use phrase:

```
use CaseRule
```

This plugin provides the following rule form:

The case condition value will be evaluated first and then all the guards value; will be evaluated in an unspecified order. Afterward, rules with a guard value equal to the value of the case condition will be evaluated. Finally, the updates generated by the matching cases are united to form the set of updates generated by the case rule.

4.7 Predicate Logic

The Predicate Logic plugin can be loaded by the following use phrase:

```
use PredicateLogic
```

This plugin provides the following functions and expression forms:

This Boolean expression holds if *guard* holds for all the elements of *value* (which must be an enumerable value).

This Boolean expression holds if there exists at least one element in value (which must be an enumerable value) that satisfies guard.

```
ightharpoonup value_1 
eq value_2 Predicate Logic Plugin
```

This is the not-equal operator which is defined on all elements.	The semantics of this operator
is equivalent to "not $(value_1 = value_2)$ ".	

▷ value₁ bin-op value₂
Predicate Logic Plugin

Performs a binary operation on the given values. The following operators are defined on Boolean values:

or, xor, and, implies

The following two operators are also defined which require $value_2$ to be an enumerable:

memberof and not memberof

▷ not value Predicate Logic Plugin

This is the negation operator which is defined on Boolean values.

4.8 Number Background

The Number plugin can be loaded by the following use phrase:

use NUMBER

This plugin provides the number background (Number) and a valuable set of functions and expression forms.

Performs binary operations on number values. Currently supported operators are

$$+ - * / div % >> = < <= =$$

which result in Number or Boolean values.

 D
 | value |
 Number Plugin

If value is enumerable (such as a set), this operator will evaluate to the size of value.

 $lack infinity: o exttt{Number}$ Number Plugin Is the positive infinity.

lacktriangledown toNumber: Element ightarrow Number Plugin

This is a conversion function that maps any value to a Number value (which can also be undef). The following example uses this function to read a number from the environment:

```
seq
  amount := input("Input Amount")
next
  let val = toNumber(amount) in
    if val = undef then
       print "Error"
  else
       DepositAmount(val)
```

- lack is Integer Number: Number o Boolean Number Plugin Returns true if the argument is an Integer number.
- lack is Real Number: Number o Boolean Number Plugin Returns true if the argument is a valid non-infinite Real number.

- lacktriangleq size: Element ightarrow Number Plugin Returns the size of the given collection.

The Number plugin also provides a background for number ranges (Number_Range). Number range elements are enumerable and can be defined using the following syntax.

Creates a range of numbers from $value_{start}$ to $value_{end}$ with the optional step. It is also possible to use ':' instead of step. In the following example, RandomGuess returns a random number between 1 and 100:

```
derived RandomGuess =
  return rand in
  choose x in [ 1 .. 100 ] do
  rand := x
```

5 Standard Plugins

Most of the CoreASM plugins, including all the Basic ASM plugins, are included in the Standard plugins package. In this section we list the plugins that are provided by the Standard plugins package in addition to the ones listed in the previous section. All these plugins can be loaded individually (as instructed in each section) or all together with the following use phrase,

```
use Standard
```

which automatically loads all the plugins listed in Section 4 in addition to the ones listed in this section.

5.1 Kernel Extensions

The Kernel Extensions plugin can be loaded by the following use phrase:

```
use KernelExtensions
```

This plugin extends the Kernel capabilities in handling function and rule elements. The current version provides the following expression and rule forms.

```
ightharpoonup id (value_1, ..., value_n) (value'_1, ..., value'_m) Kernel Extensions Plugin 
ightharpoonup (value'_1, ..., value'_m) Kernel Extensions Plugin
```

The above two forms apply the arguments $value^{i}$ to the function element at location (value) ($value_{1}$, ..., $value_{n}$) or to the function element resulting from evaluation of value. If the function element refers to a function in the state, the location of the above expressions are also set to the location of the function with the given arguments; otherwise (e.g., in case of non-state functions) the location will be not be defined. Here are some examples, assuming that foo and value0 defined functions, and value1 defined functions.

```
print bar()(5,4) //printing the value of foo(5, 4) (bar)(1, 3) := 4 //assigning 4 to foo(1, 3)
```

```
lackbox{ call } id \ (value_1, \ ..., \ value_n) \ \ (value'_1, \ ..., value'_m) \ \ \ ... \ \ \ \  Kernel Extensions Plugin
```

```
lacktriangledown call (value) (value'_1,\ ...,\ value'_m) . . . . . . . . . . . . Kernel Extensions Plugin
```

The above two rules call the rule element value of id ($value_1,..., value_n$) (the first form) or value (the second form) with the arguments $value'_i$. For example, if we have foo(5) = @MyRule and

then we can call this rule by:

```
call foo(5) ("John", "Mary") // prints "John talks to Mary"
```

This plugin is not yet part of the Standard Plugin package.

5.2 Abstraction

The Abstraction plugin can be loaded by the following use phrase:

```
use Abstraction
```

This plugin provides the following rule form, which is useful when the specifier wants to leave the detail of a rule abstract.

In the following example, the rule SendMessage is left abstract:

```
rule SendMessage =
   abstract "Sending the message."
```

5.3 Extend Rule

The Extend Rule plugin can be loaded by the following use phrase:

```
use ExtendRule
```

This plugin provides the following rule form:

▶ extend value with id do rule Extend Rule Plugin

This rule has two semantics depending on value:

- 1. If value is a universe, it imports a new element, assigns it to id, and evaluates rule. The resulting update set is the union of the updates generated by rule and a single update to add the imported element to the universe value.
- 2. If *value* is a background, it gets the default element from the background, assigns it to *id* and evaluates *rule*. The resulting update set is the updates generated by *rule*.

In the following example, the universe AGENTS is extended with a new agent and the program of that agent is set to MainProgram:

```
extend AGENTS with a do
program(a) := @MainProgram
```

However, the same result can be achieved by:

```
import a do
   par
   AGENTS(a) := true
   program(a) := @MainProgram
endpar
```

5.4 TurboASM Rules

The TurboASM plugin can be loaded by the following use phrase:

```
use TurboASM
```

This plugin provides the following rule forms:

```
lackbox{f seq} \ rule_1 \ {f next} \ rule_2 \ {f next} \ rule_3 \ ... \ {f next} \ rule_n \ {f optional} \ {f optiona
```

Evaluates $rule_1$, applies the generated updates in a virtual state, and evaluates $rule_2$ in that state. The resulting update set is a sequential composition of the updates generated by $rule_1$, $rule_2$, and all other rules $rule_n$. The keyword next is meant to improve readability specially where the sequence rule is combined with other rule forms. In order to avoid ambiguities, the optional keyword endseq can be used to explicitly complete a seq ... next group.

```
      ▶ seqblock rule_1 ... rule_n endseqblock
      TurboASM Plugin

      ▶ seq rule_1 ... rule_n endseq
      TurboASM Plugin
```

```
lackbox{ } lac
```

Similar to the **seq** rule (above), these block rules execute the contained rules in sequence. First, $rule_1$ is evaluated and the generated updates are applied to a virtual state. This state is the base for the evaluation of $rule_2$ which may produce further updates to this virtual state, and so on. The resulting update set is a sequential composition of the updates generated by $rule_1 \dots rule_n$.

Repeatedly evaluates *rule*, until the update set produced is either empty or inconsistent; at that point, the accumulated updates are computed (the resulting update set can be inconsistent if the computation of the last step had produced an inconsistent set of updates).

This rule is equivalent to:

```
iterate
  if value then rule
```

 $lackbox{ loc <- rule}$ TurboASM Plugin

Replaces all the occurrences of **result** in *rule* with *loc* and evaluates the rule. In the ASM book this is written as "*loc* <- *rule*". In the following example, the evaluation of MainProgram assigns the value of 5 divided by 2 (i.e., 2.5) to division:

```
rule Divide(a, b) =
  if b > 0 then
    result := a / b
  else
    par
    result := undef
    error := true
    endpar

rule MainProgram =
  division <- Divide(5, 2)</pre>
```

First, *rule* is evaluated; *value* is then evaluated in the state obtained by provisionally, and the *value* is returned, while the updates and the provisional state itself are discarded.

Remark The return-construct has been changed from a rule-construct to an expression-construct. This decision has been taken in order to clarify the roles of derived function and rules. Now, after removing "return rules" all macro rules in principal have side-effects and only derived functions are side-effect-free by definition.

Evaluates rule but discards all the updates to locations addressed by id-s (as location names). In the following example, newValue will get the local value of foo(5, 7) (i.e., 25) but the update to foo(5, 7) will be discarded afterwards.

```
rule LocalRule =
  local foo in
  seq
  foo(5, 7) := 25
  newValue := foo(5, 7)
```

5.5 String Background

The String plugin can be loaded by the following use phrase:

```
use STRING
```

This plugin provides the string background (STRING) and few functions and expression forms.

```
▷ value<sub>1</sub> + value<sub>2</sub> String Plugin
```

If both values are string, this operator concatenates the given string values in to one. If one of the values is not a string value, it tries to convert it into a string value, and then concatenates the values. This operator is not defined on two non-string values.

With this operator, one can simply put values together to create a customized message:

```
| print "The amount of $" + amount + " is deposited to your account."
```

lack to String: Element o String Plugin

A conversion function that maps any value to a String value (which can also be undef).

- ♦ strlen: String → Number String Plugin Returns the length of the given String value.
- ♦ matches: STRING → STRING String Plugin

Returns **true**, if the first parameter matches the given regular expression provided by the second parameter. Otherwise **false** is returned. The syntax for the regular expressions follows the java language definition. For example, the function *matches*("42", "[0-9]+") returns **true**.

5.6 Input and Output

The IO plugin can be loaded by the following use phrase:

use IO

This plugin provides the following rule form and function which enable user input and output as well as file input and output:

Remark All values that describe file destinations are terms from the String background. A file destination can be either described relatively to the executed specification, i. e. the main specification that is executed by CoreASM (not necessary the module which contains the print-to-file construct), e. g. "./", or the file destination is described relatively to the systems root folder, e. g. "c:" or "/"

▶ print valueIO Plugin

Prints out value to the environment. Depending on the environment (engine driver) this value can be printed on the standard output.

Prints $value_1$ into a new file named $value_2$, if consistent. If it does not exist, this file will be created. If it already exists it is overwritten without any further warning. Instead of the keyword to, maybe some linux users prefer the operator > (which can be used, too).

Prints $value_1$ into an existing file named $value_2$, if consistent. If the file already exists the value is appended to the existing content of the file. If it does not exist, this file will be created. The alternative operator \gg can be used Instead of the keyword into.

Reads a string value from the environment. Given a step and given an argument arg, every evaluation of *input*(arg) during this step will result in the same value. Please refer to Section 5.5 for an introduction to the String Plugin.

The derived function read(value) returns the content from the given filename as List of elements from the String Background. The returned list contains the lines from the source file in ascending order.

The machine specified in CoreASM-Says-Hello example with IO extension is an extension of our CoreASM-Says-Hello example that reads a name from the environment and prints out a greeting to that name:

```
CoreASM-Says-Hello example with IO extension
   CoreASM ThisIsCoreASM
   use Standard
   init InitRule
   rule InitRule =
      par
          terminate := false
          program(self) := @MainProgram
         name := input("What is your name?")
       endpar
   rule MainProgram =
       if not terminate then
         par
             print "This is CoreASM."
             terminate := true
             print "Hello " + name + "!"
          endpar
          program(self) := undef
```

5.7 Collection

The Collection plugin can be loaded by the following use phrase:

```
use Collection
```

This plugin provides the foundation for collections (i.e., sets, lists, maps, etc.) in CoreASM and provides some general functions on collections. However, each specific collection background (e.g., list or set) is provided by its corresponding plugin.

foldl(c, @func, init) processes the collection c (e.g., a set or a list) using the binary function func and the initial value init and returns the final result.

$$foldl([x_1,\ldots,x_n],f,i) \equiv f(x_n,f(x_{n-1},\ldots,f(x_1,init)))\ldots)$$

foldr(c, @func, init) processes the collection c (a set or a list) using the binary function func and the initial value init and returns the final result.

$$foldr([x_1,\ldots,x_n],f,i) \equiv f(x_1,f(x_2,\ldots,f(x_n,init)))\ldots)$$

$$map([x_1,...,x_n], f) \equiv [f(x_1), f(x_2),...f(x_n)]$$

filter(c, @func) applies the boolean unary function func to all the elements of c and returns a new collection with only those elements of c for which func returns true.

5.8 Set Background

The Set plugin can be loaded by the following use phrase:

use SET

This plugin provides the set background (SET) and a number of functions and expression forms.

$$ightharpoonup \left\{ \begin{array}{ll} \underline{\mathit{value}_1, \ ..., \ \mathit{value}_n} \end{array}
ight\}$$
 Set Plugin $\underbrace{\mathit{optional}}$

Creates a set element that includes the listed values. The values should be basic terms (i.e., no operators) or they should be surrounded in parentheses.

```
ightharpoonup \{ id \ | id \ in \ value \ rac{	ext{with} \ guard}{optional} \} Set Plugin
```

This is the basic form set comprehension. It creates a set of all the elements in value which satisfy guard. Of course, value must be enumerable.

```
ho = \{id \text{ is } exp \mid id_1 \text{ in } value_1, \dots, id_n \text{ in } value_n = with \ guard \ \} Set Plugin optional
```

Creates a set element that contains all the elements of the form exp which satisfy the guard. In this form, exp is a function of $id_1, ..., id_n$ and every id_i is bound to an enumerable $value_i$. In the following example, SetAdd takes two sets set1 and set2 as input and produces a new set by adding every element of set1 to all the elements of set2:

```
derived SetAdd(set1, set2) =
   return a in
   a := { x is (x1 + x2) | x1 in set1, x2 in set2 }
```

The result of evaluating $SetAdd(\{1, 2, 3\}, \{10, 20\})$ would be:

```
{22.0, 23.0, 12.0, 21.0, 13.0, 11.0}
```

Performs a set binary operation where both $value_1$ and $value_2$ are sets. Currently, subset, union, intersect, and diff are supported.

Set background also provides two important rule forms which allow for parallel incremental updates of set data structures.

```
▶ add value to loc Set Plugin
```

If loc is a location in the state (e.g., a function) and its value is a set, this rule produces an update instruction (partial update) that adds value to loc.

If *loc* is a location in the state (e.g., a function) and its value is a set, this rule produces an update instruction (partial update) that removes *value* to *loc*.

5.9 Bag Background

The Bag plugin can be loaded by the following use phrase:

```
use Bag
```

This plugin provides the bag background (Bag) equivalent to multi sets and a number of functions and expression forms.

$$ightharpoonup < rac{value_1, \ ..., \ value_n}{optional}
ightarrow
ightharpoonup$$
 Bag Plugin

Creates a bag element that includes the listed values. The values should be basic terms (i.e., no operators) or they should be surrounded in parentheses.

$$ightharpoonup <<< id | id in value with guard >> Bag Plugin optional Bag Plugin Bag Plugin$$

This is the basic form bag comprehension. It creates a bag of all the elements in *value* which satisfy *guard*. Of course, *value* must be enumerable.

Performs a bag binary operation where both *value*₁ and *value*₂ are bags. Currently, *subset*, *union*, *intersect*, and *diff* are supported.

Creates a bag element that contains all the elements of the form exp which satisfy the guard. In this form, exp is a function of $id_1, ..., id_n$ and every id_i is bound to an enumerable $value_i$. In the following example, BagAdd takes two bags bag1 and bag2 as input and produces a bag that contains all elements of bag1 and all elements of bag2:

```
derived BagMerge(bag1, bag2) = bag1 + bag2
```

The result of evaluating BagAdd(<<1, 2, 3>>, <<2, 3, 3>> would be:

```
<< 1, 2, 2, 3, 3, 3 >>
```

Bag background also provides two important rule forms which allow for parallel incremental updates of bag data structures.

```
► add value to loc Bag Plugin
```

If loc is a location in the state (e.g., a function) and its value is a bag, this rule produces an update instruction (partial update) that adds value to loc.

► remove value from loc Bag Plugin

If loc is a location in the state (e.g., a function) and its value is a bag, this rule produces an update instruction (partial update) that removes value to loc.

5.10 List Background

The List plugin can be loaded by the following use phrase:

use LIST

This plugin provides a list background (List) and a rich set of functions and operators on lists.

 $ightharpoonup \left[\begin{array}{c} value_1, \ value_2, \ ..., \ value_n \end{array} \right]$ List Plugin

Creates a list element that includes $value_1$ to $value_n$ in the given order.³ List elements are enumerable. The index of the first element is 1.

ho value_1 + value_2 List Plugin

If both values are list, this operator concatenates the given lists in to one list.

lack to List: Element o List Plugin

If e is an enumerable (e.g., number range, set, etc.), to List(e) will return a list that includes all the elements of e. If e is not ordered (e.g., a set), the order of elements in the returned list will be non-deterministic; otherwise the elements will be in the same order.

- lacktriangledaws flattenList: LIST ightarrow List Plugin

 If 1 is a netsting list, flattenList(1) will return a flatten version of 1.
- lacktriangleq head: List ightarrow Element ightarrow List Plugin Returns the first element of the list.
- ♦ $last: List \rightarrow Element$ List Plugin

 $^{^3}$ The old form of <<x1,... ,xn>> still works but it is deprecated and may not be supported in future releases of the CoreASM engine.

Returns the last element of the list.

lacktriangledown $tail: List o Li$
Returns all but the first element of the list.
lacktriangledown cons: Element $ imes$ List $ o$ List Plugin
Creates a new list with the given element as its head and given list as its tail.
Returns the n^{th} element of the list. The index of the first element is 1.
$setnth(\mathtt{list},\ \mathtt{i},\ \mathtt{e}),\ \mathrm{if}\ \mathtt{i}\ \mathrm{is}\ \mathrm{a}\ \mathrm{valid}\ \mathrm{index}\ \mathrm{for}\ \mathtt{list},\ \mathrm{returns}\ \mathrm{a}\ \mathrm{new}\ \mathrm{list}\ \mathrm{in}\ \mathrm{which}\ \mathrm{the}\ \mathrm{element}\ \mathrm{at}$ index $\mathtt{i}\ \mathrm{is}\ \mathtt{e}.$
$lacktriangledown$ $take: List imes exttt{Number} o List exttt{Plugin}$
take(list, i) returns the first i elements of list list.
$drop(\mathtt{list},\ \mathtt{i})$ returns what is left after dropping the first i elements of the list $\mathtt{list}.$
lacktriangledown reverse: List $ ightarrow$ List Plugin
Returns a list consisting of the given list's elements in reverse order.
lacktriangledown indexes: List $ imes$ Element $ o$ List Plugin
Returns a potentially empty list of the indexes of the given element in given list.
$ \blacklozenge \ \ indices: \ List \times \ Element \ \to \ List \ Plugin $
The same as indexes; see above.
lacktriangledown $zip:$ LIST $ imes$ LIST $ o$ LIST
The function zip takes two lists and returns a list of corresponding pairs. If one input list is short, excess elements of the longer list are discarded.
lacktriangledown zipwith: List $ imes$ List $ imes$ Function $ o$ List List Plugin

The function *zipwith* generalises zip by zipping with the function given as the last argument, instead of a tupling function. For example, *zipwith* (11, 12, @max) is applied to two lists to produce a list of corresponding maximums (requires use Math).

lack replicate: Element imes Number o List Plugin

The function replicate(x, n) returns a new list where the given element x is repeated n times.

List background also provides the following rule forms to manipulate lists:

```
► add value to loc List Plugin
```

If loc is a location in the state and its value is a list, this rule produces an update that adds value to loc. In lists order matters, so the update produced by this rule is NOT incremental (not like the one for sets). As a result, there cannot be two parallel add rules operating on the same list.

```
► remove value from loc List Plugin
```

If *loc* is a location in the state and its value is a list, this rule produces an update that removes the first occurrence of *value* from *loc*. As for add, this rule is also NOT incremental (not like the one for sets) and there cannot be two parallel **remove** rules operating on the same list.

```
► shift left value into loc List Plugin
```

If loc is a location in the state and value is a list, it removes the first element of the list and puts it in the given location (shifting the list to left).

```
► shift right value into loc List Plugin
```

If loc is a location in the state and value is a list, it removes the last element of the list and puts it in the given location (shifiting the list to right).

In the following example, SortSet sorts elements of a given set into a list:

```
rule SortSet(set) =
    seq
    par
        result := []
        tempSet := set
    endpar
    next
    while ( | tempSet | > 0 )
        choose x in tempSet with forall y in tempSet holds x <= y do
        par
            remove x from tempSet
        add x to result</pre>
```

endpar

5.11 Queue

The Queue plugin can be loaded by the following use phrase:

use Queue

This plugin provides the following queue operations (rule forms) on lists:

If loc is a location in the state and its value is a queue (i.e., a list), it adds value to the end of the queue.

If loc_q is a location in the state and its value is a queue (i.e., a list), it removes the first element of this queue and assigns it as the value of the location loc_v .

5.12 Stack

The Stack plugin can be loaded by the following use phrase:

use Stack

This plugin provides the following stack operations and functions on lists:

If loc is a location in the state and its value is a stack (i.e., a list), it pushes value to the front of the stack.

If loc_s is a location in the state and its value is a stack (i.e., a list), it removes the first element of the stack (top of the stack) and assigns it as the value of loc_v .

Returns the top of the stack (first element of the list) without changing the stack.

5.13 Map Background

The Map plugin can be loaded by the following use phrase:

```
use MAP
```

This plugin provides a map background (MAP).

```
\, \, \triangleright \,\, \left\{ \, \rightarrow \, \right\} \hspace{1cm} \mathsf{Map \, Plugin}
```

Creates an empty map.

```
ho { value_{k1} 	o value_{v1}, value_{k2} 	o value_{v2}, \dots, value_{kn} 	o value_{vn}} } \cdots Map Plugin optional
```

Creates a map with the given key-value pairs. Map elements are enumerable; every map can be viewed as a set of pairs which are represented by lists of size 2.

```
♦ toMap: Element \rightarrow Map
```

If e is an enumerable (e.g., a set) consisting of pairs of elements (lists of size two) of the form $[k_i, v_i]$ such that $\forall [k_i, v_i] \ / \exists [k_j, v_j] \ k_i = k_j \land v_i \neq v_j$, toMap(e) returns a map element representing a mapping of k_i s to v_i s; otherwise, it returns undef. For example, the following two expressions create equal maps:

```
toMap(\{[1, "John"], [2, "Mary"]\})
results in
\{1 \rightarrow "John", 2 \rightarrow "Mary"\}
```

```
lack mapToPairs: MAP 
ightarrow SET Map Plugin
```

Returns a set of pairs of the form (key, value) from the given map elements. The pairs are list elements of size two. For example, the following two expressions are equal:

```
mapToPairs(\{1 \rightarrow "John", 2 \rightarrow "Mary"\}) results\ in \{[1, "John"], [2, "Mary"]\}
```

Map background also provides the following rule forms to manipulate maps:

If *loc* is a location in the state, its value is a map, and *value* is a map, this rule produces an update that copied all of the mappings from *value* to *loc*. These mappings will replace any mappings that *loc* had for any of the keys shared with *value*. In the current version of Map plugin, the update produced by this rule is NOT incremental (not like the one for sets). As a result, there cannot be two parallel add rules operating on the same map.

If loc is a location in the state and its value is a map, this rule produces an update that removes value from loc according to the following:

- 1. if value is a map, this rule removes all the exact mappings of value from loc;
- 2. if *value* is not a map but an enumerable, this rule removes all the mappings for the elements of *value* (as keys) from *loc*;
- 3. if *value* is neither a map nor an enumerable, this rule removes the mapping for *value* (as a key) from *loc* if present.

In the current version of Map plugin, the update produced by this rule is NOT incremental (not like the one for sets). As a result, there cannot be two parallel **remove** rules operating on the same map.

5.14 Signature Plugin

The Signature plugin can be loaded by the following use phrase:

```
use Signature
```

The signature plugin extends the header section of CoreASM specifications (see Figure 1) to add support for definition of functions, universes, and custom data types and also extends the engine to support for certain forms of type checking. This plugin is still under development. The current version includes the following features:

• Definition of universes through the following syntax (with optional initial elements):

$$\begin{array}{c} \textbf{universe} \hspace{0.1cm} \textit{id} \hspace{0.1cm} \underline{= \hspace{0.1cm} \{ \hspace{0.1cm} \textit{id}_1, \hspace{0.1cm} ..., \hspace{0.1cm} \textit{id}_n \}} \\ \hspace{0.1cm} \textit{optional} \end{array}$$

• Definition of enumeration backgrounds through the following syntax:

$$egin{aligned} extbf{enum} & id = \{ & id_1, & ..., & id_n \} \ & & optional \end{aligned}$$

For example, the following line defines a new enumeration background of four elements:

```
enum \ Product = \{ \ Soda, \ Juice, \ Sandwich, \ Candy \}
```

The elements are in fact defined as constant functions that hold values of the background Product.

• Definition of functions through the following syntax:

As an example, the following signature defines a function named priceTable that maps pairs of string values to numbers:

```
| function priceTable : STRING * STRING → NUMBER
```

• Definition of derived functions through the following syntax:

$$\begin{array}{c} \textbf{derived} \ \ id_f \ \underline{(\ id_1, \ ..., \ id_n)} = \ expression \\ \hline \ \ optional \end{array}$$

As an example, the following declaration defines a derived function $f(x,y) = x^2 + y^2$:

$$| derived f(x, y) = x^2 + y^2$$

Depending on the properties of the engine (see the Options Plugin, Section 6.2) the Signature plugin can use the signature information to perform the following checks:

- Type checking on assignments: if the "Signature.TypeChecking" property is set to "warning", "strict" or "on", before the updates are applied to the state, the Signature Plugin checks the types of arguments and assigned values against the defined signatures and issues a warning (in case of "warning") or stops the execution of the engine with an error (in case of "strict" or "on").
- Unknown identifiers: if the "Signature.NoUndefinedId" property is set to "warning", "strict" or "on", the Signature Plugin issues a warning (in case of "warning") or stops the execution of the engine with an error (in case of "strict" or "on") if a function name is used and its signature is not defined in the header of the specification. This feature helps in identifying typos in the specification.

6 Additional Plugins

The plugins listed in this section are currently not part of any plugin packages.

6.1 Modularity

The Modularity plugin can be loaded by the following use phrase:

```
use Modularity
```

This plugin allows one to break the specification into separate files or *modules*. As its current version, the functionality provided is limited to introducing an **include** keyword that would load another file into the current specification.

```
include "filename"
```

Included files can themselves have other **include** clauses to further break down the specification.

6.2 Options

The Options plugin can be loaded by the following use phrase:

```
use Options
```

The Options plugin extends the header section of CoreASM specifications (see Figure 1) to provide the following syntax to set values of engine properties:

```
option property value
```

Other plugins (such as the Signature Plugin, see Section 5.14) can use these options to customize their behavior.

6.3 Scheduling Policies

The Scheduling Policies plugin can be loaded by the following use phrase:

```
use SchedulingPolicies
```

This plugin provides alternative scheduling policies for simulation of multi-agent specifications. For any specification (for any run), only one scheduling policy can be defined, using the following option:

```
option SchedulingPolicies.policy policyname
```

Currently, there are two scheduling policies provided by this plugin:

- allfirst Tries executing all the agents in every computation step. If this fails at any step, the policy falls back to the engine's default scheduling policy.
- onebyone Executes only one agent in every step. It tries to be fair by not executing an agent more than once unless all other agents have been given a chance to execute.

The following rules are also provided by this plugin to control the execution of agents during a simulation.

► suspend value SchedulingPolicies Plugin

If *value* is an agent, this rule suspends the execution of that agent from the next computation step. The suspended agents will not be chosen by the engine for execution.

► resume value SchedulingPolicies Plugin

If value is an agent which has been suspended, this rule resumes the execution of that agent from the next computation step; i.e., the agent will be available for execution from the next step.

► terminate value SchedulingPolicies Plugin

If value is an agent, it will no longer be available for scheduling for the rest of the current run of the machine.

▶ shutdown SchedulingPolicies Plugin

Clears the AGENTS universe, such that there will be no agent available to contribute to the next computation step. Depending on the parameters of the run, this can stop the execution of the engine.

6.4 Time

The Time plugin can be loaded by the following use phrase:

use Time

This plugin provides the following monitored function:

♦ now: → Number Time Plugin

Returns a value representing the current time of the system. Of course, given a step, the value of now is fixed.

Returns the number of computation steps performed so far by the engine excluding the current step.

6.5 DebugInfo

DebugInfo plugin is a CoreASM plugin to maintain logging information for debugging purposes and it can be loaded by the following use phrase:

```
use DebugInfo
```

The plugin adds the following rule to the CoreASM language:

which, upon evaluation, adds the string representation of the given value to the logging channel identified by the given id.

The set of active channels are to be defined as a space-separated list of channel ids, set as the value <code>DebugInfo.activeChannels</code> engine property. This can be done either through the Options plugin or by setting the values directly from the engine driver (e.g., <code>Carma</code>). For example, using the Options plugin one can add the following line to a spec to turn the logging on for channels <code>warning</code> and <code>error</code>:

```
option DebugInfo.activeChannels "warning, error"
```

In order to turn all channels on, one can use the special channel id ALL:

```
option DebugInfo.activeChannels ALL // or "ALL"
```

Since this rule is only used for debugging purposes, the evaluation of *debuginfo* results in an empty update set and a print out of the debugging information (if the corresponding channel is active) to the standard output, whether or not the updates of the enclosing rule block is discarded by the engine or not. Applications of the engine can set redirect the output of this plugin using the plugin's service interface (see org.coreasm.engine.plugin.Plugin#getPluginInterface()).

Example

```
CoreASM DebugInfoExample

use Standard
use DebugInfo
use Options

option DebugIinfo.activeChannels ALL
```

```
//option DebugIinfo.activeChannels "ch1 ch2"
//option DebugIinfo.activeChannels "ch1, ch2"
// {\rm option\ Debug Iinfo.active Channels\ ch1}
//option DebugIinfo.activeChannels NONE
init R1
rule R1 =
   if mode = undef then
      par
          debuginfo ch1 "initializing."
          mode := "counting"
          counter := 0
      endpar
   else
      par
          debuginfo ch2 mode
          counter := counter + 1
      endpar
```

6.6 Math

The Math plugin can be loaded by the following use phrase:

```
use Math
```

Math Plugin extends the CoreASM engine to provide some basic mathematical functions. Most of these functions are equivalent of their Java counterparts in java.lang.Math. For such functions, the following descriptions are basically taken from the Java 2 Platform Standard Edition 5.0 API Specification.

6.6.1 Constants

MathE

The value that is closer than any other to e, the base of the natural logarithms.

MathPI

The value that is closer than any other to π , the ratio of the circumference of a circle to its diameter.

6.6.2 Basic Derived Functions

- abs(v) Returns the absolute value of v.
- acos(v) Returns the arc cosine of an angle, in the range of 0 through π .
- asin(v) Returns the arc sine of an angle, in the range of $-\pi/2$ through $\pi/2$.

- atan(v) Returns the arc tangent of an angle, in the range of $-\pi/2$ through $\pi/2$.
- atan2(x, y) Converts rectangular coordinates (x, y) to polar (r, θ) and returns θ .
- cuberoot(v) Returns the cube root of v.
- cbrt(v) Returns the cube root of v.
- ceil(v) Returns the smallest (closest to negative infinity) value that is greater than or equal to the argument and is equal to a mathematical integer.
- cos(v) Returns the trigonometric cosine of an angle.
- cosh(v) Returns the hyperbolic cosine of v.
- exp(v) Returns Euler's number e raised to the power of v.
- expm1(v) Returns $e^v 1$.
- floor(v) Returns the largest (closest to positive infinity) value that is less than or equal to the argument and is equal to a mathematical integer.
- hypot(x, y) Returns $\sqrt{x^2 + y^2}$ without intermediate overflow or underflow.
- IEEEremainder(v1, v2) Computes the remainder operation on two arguments as prescribed by the IEEE 754 standard.
- log(v) Returns the natural logarithm (base e) of v.
- log10(v) Returns the base 10 logarithm of v.
- log1p(v) Returns the natural logarithm of the sum of the argument and 1; i.e., ln(v+1).
- max(v1, v2) Returns the greater of two values.
- min(v1, v2) Returns the smaller of two values.
- \bullet pow(x, y) Returns the value of the first argument raised to the power of the second argument.
- random() Returns a random value with a positive sign, greater than or equal to 0.0 and less than 1.0.
- round(v) Returns the closest mathematical integer to the argument.
- signum(v) Returns zero if the argument is zero, 1.0 if the argument is greater than zero, -1.0 if the argument is less than zero.
- sin(v) Returns the trigonometric sine of an angle.
- sinh(v) Returns the hyperbolic sine of v.
- sqrt(v) Returns the correctly rounded positive square root of v; i.e., \sqrt{v} .

- tan(v) Returns the trigonometric tangent of an angle.
- tanh(v) Returns the hyperbolic tangent of v.
- toDegrees(v) Converts an angle measured in radians to an approximately equivalent angle measured in degrees.
- toRadians(v) Converts an angle measured in degrees to an approximately equivalent angle measured in radians.

6.6.3 Special Derived Functions

- powerset(set) Computes the powerset of the given set.
- powerset({e1,...,en}) This function returns the powerset of the given set of elements.
- max({v1,...,vn}) Returns the maximum value in a collection of numbers. If there is one non-number in the collection, it returns undef.
- min({v1,...,vn}) Returns the minimum value in a collection of numbers. If there is one non-number in the collection, it returns undef.
- sum({v1,...,vn}) This function returns the sum of a collection of numbers. If there is one non-number in the collection, it returns undef.
- sum({v1,...,vn}, @f) This function returns the sum of a collection of numbers, after applying function f to the values in the collection. If there is one non-number in the collection, it returns undef.

6.6.4 An Example

```
Using Math Plugin
    CoreASM MathPluginExample
    use StandardPlugin
    use MathPlugin
    init Init
    rule Init =
      par
          program(self) := @Main
          a(1) := 5
          a(2) := 10
          a(100) := 500
       endpar
    rule Main =
       let e = MathE in
          par
             print "'e' = " + e
             print "log(e) = " + log(e)
             print "sin(30) = " + round( sin( toRadians(30) ) * 10 ) / 10
             print "asin(0.5) = " + round( to Degrees( asin(0.5) ))
             print "min(51, 43) = " + min(51, 43)
             print "sum(\{1, 2, 100\}) = " + sum(\{1, 2, 100\})
             print "sum( {1, 2, 100}, @a ) = " + sum({1, 2, 100}, @a)
             print "\{2, 3\} is in P(\{1, 2, 3\}) = " + (\{2, 3\} \text{ memberof})
        powerset({1,2,3}))
             choose x in powerset(\{1, 2, 3, 4\}) do
                if x memberof powerset({1, 2, 3}) then
                   print x + " is a member of powerset({1, 2, 3})"
                   print x + " is not a member of powerset({1, 2, 3})"
          endpar
```

As an example, the output of the CoreASM Spec MathPluginExample would be the following:

```
 \begin{aligned} & \text{sum}(\ \{1,\ 2,\ 100\}\ ) \ = \ 103 \\ & \text{min}(51,\ 43) \ = \ 43 \\ & \text{asin}(0.5) \ = \ 30 \\ & \text{powerset}(\{1,\ 2,\ 3\}) \ = \ \{\{\},\ \{3\},\ \{2\},\ \{3,\ 2\},\ \{1\},\ \{3,\ 1\},\ \{2,\ 1\},\ \{3,\ 2,\ 1\}\} \\ & \{2,\ 3\} \ \text{memberof powerset}(\{1,\ 2,\ 3\} \ = \ \text{true} \\ & \log(e) \ = \ 1 \\ & \text{sum}(\ \{1,\ 2,\ 100\},\ @a\ ) \ = \ 515 \\ & \text{'e'} \ = \ 2.718281828459045 \\ & \{2,\ 1,\ 4\} \ \text{is not a member of powerset}(\{1,\ 2,\ 3\}) \\ & \sin(30) \ = \ 0.5 \end{aligned}
```

7 Notes about the CoreASM Compiler

As mentioned in section 1.3, the CoreASM Compiler currently does not provide support for all CoreASM Plugins. Supported are:

- All Standard plugins
 - BlockRulePlugin
 - ChooseRulePlugin
 - ConditionalRulePlugin
 - ExtendRulePlugin
 - ForallRulePlugin
 - IOPlugin
 - LetRulePlugin
 - NumberPlugin
 - PredicateLogicPlugin
 - SetPlugin
 - SignaturePlugin
 - StringPlugin
 - TurboASMPlugin
 - CollectionPlugin
 - ListPlugin
 - MapPlugin
 - AbstractionPlugin
 - CaseRulePlugin
 - OptionsPlugin

- KernelExtensionsPlugin
- MathPlugin
- ModularityPlugin
- TimePlugin

Still, some restrictions apply to several of the mentioned plugins.

Kernel The MacroCall operation has some slight differences between the interpreter and the compiler versions. They shouldn't influence a well written specification, but can still provide errors.

SignaturePlugin The SignaturePlugin provides an undef-handler to the CoreASM engine, which allows to generate warnings and errors upon using undefined locations. This currently doesn't work in the compiler.

TurboASMPlugin The TurboASMPlugin Return Result rule might not work as intended in all instances, but should provide the same result as the CoreASM Engine in most cases.

KernelExtensionsPlugin The KernelExtensionsPlugin is only implemented partially, missing some functionality.

Index

:=, see update rule	false, 10
<- rule, see return result rule	filter function, 25
=, see equality operator	fold function, 25
x , see size-of operator	foldl function, 24
[], see list element, see map element	foldr function, 25
{}, see set element, see bag element	forall rule, 14
[], see number range elements	Forall Rule plugin, 14
1, see number range elements	function, 34
Abstraction plugin, 19	Tunction, 94
AGENTS, 10	Header block, 9
and, see Boolean operators	
and, see Boolean operators	if-then-else rule, see conditional rule
Bag background, 27	implies, see Boolean operators
bag comprehension, 27	import, 11
bag element, 27	include, 35
bag enumeration, 27	infinity function, 16
Bag plugin, 27	init, see init rule
Basic ASM plugin, 12	init rule, 9
block rule, 12	input function, 23
Block Rule plugin, 12	IO plugin, 23
Boolean background, 10	isEvenNumber function, 17
Boolean operators, 16	isIntegerNumber function, 17
Boolean operators, 10	isNaturalNumber function, 17
case rule, 15	isOddNumber function, 17
Case Rule plugin, 15	isRealNumber function, 17
choose rule, 13	isiteanvamoer rancolon, 17
Choose Rule plugin, 13	kernel, see CoreASM kernel
Collection plugin, 24	Kernel Extensions plugin, 18
conditional operation, 14	1 0 /
conditional rule, 14	let rule, 15
Conditional Rule plugin, 13	Let Rule plugin, 14
controlled, 34	List background, 28
CoreASM, 7	list concatenation, 28
CoreASM kernel, 10	list element, 28
	List plugin, 28
derived, 34	
	Map background, 32
enum, see enumeration background	map element, 32
enumerable, 13	map function, 25
enumeration backgrounds, 33	Map plugin, 32
equality operator, 11	matches function, 22
extend rule, 19	Math plugin, 38
Extend Rule plugin, 19	memberof operator, see membership operators
	membership operators, 16

Modularity plugin, 35
not operator, 16 not-equal operator, 15 now function, 36 Number background, 16 Number plugin, 16 Number Range background, 17 number range elements, 17
option, 35 Options plugin, 35 or, see Boolean operators
par, see block rule Predicate Logic plugin, 15
queue, 31 Queue plugin, 31
read function, 23 return, 21 return result rule, 21 Rule Declaration, 9 ruleelement, 11
Scheduling Policies plugin, 35 self, 10
seqblock, see sequence block rule seq rule, 20 sequence block rule, 20, 21 Set background, 25 set comprehension, 26 set element, 25, 26 set enumeration, 25 Set plugin, 25 Signature plugin, 33 size function, 17 size-of operator, 16 stack, 31 Stack plugin, 31 Standard plugins, 18 stepcount function, 36 String background, 22 string concatenation, 22 String plugin, 22

```
Time plugin, 36

toNumber function, 16

toString function, 22

true, 10

TurboASM plugin, 20

undef, 10

universe, 33

update rule, 11

use, 9

xor, see Boolean operators
```