

# CoreASM Language User Manual

engine version 1.6.4-beta

[github.com/CoreASM/](https://github.com/CoreASM/)

Roozbeh Farahbod  
info@coreasm.org

Marcel Dausend  
marcel.dausend@uni-ulm.de

Copyright © 2006-2014

**DRAFT of August 20, 2014**

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

## Contents

<b>1</b>	<b>Installing CoreASM</b>	<b>4</b>
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 . . . . .	4
1.2.3	Using CoreASM Eclipse Plugin . . . . .	5
<b>2</b>	<b>CoreASM Specification</b>	<b>6</b>
2.1	Running CoreASM Specifications . . . . .	7
<b>3</b>	<b>Kernel</b>	<b>8</b>
3.1	Rule Forms . . . . .	9
3.2	Kernel Engine Properties . . . . .	9
<b>4</b>	<b>Basic ASM Plugins</b>	<b>10</b>
4.1	Block Rule . . . . .	10
4.2	Choose Rule . . . . .	11
4.3	Conditional Rule . . . . .	11
4.4	Forall Rule . . . . .	12
4.5	Let Rule . . . . .	12
4.6	Case Rule . . . . .	13
4.7	Predicate Logic . . . . .	13
4.8	Number Background . . . . .	14
<b>5</b>	<b>Standard Plugins</b>	<b>16</b>
5.1	Kernel Extensions . . . . .	16
5.2	Abstraction . . . . .	17
5.3	Extend Rule . . . . .	17
5.4	TurboASM Rules . . . . .	18

5.5	String Background . . . . .	20
5.6	Input and Output . . . . .	21
5.7	Collection . . . . .	22
5.8	Set Background . . . . .	23
5.9	List Background . . . . .	24
5.10	Queue . . . . .	27
5.11	Stack . . . . .	28
5.12	Map Background . . . . .	28
5.13	Signature Plugin . . . . .	30
<b>6</b>	<b>Additional Plugins</b>	<b>31</b>
6.1	Modularity . . . . .	31
6.2	Options . . . . .	31
6.3	Scheduling Policies . . . . .	32
6.4	Time . . . . .	33
6.5	DebugInfo . . . . .	33
6.6	Math . . . . .	34
6.6.1	Constants . . . . .	35
6.6.2	Basic Derived Functions . . . . .	35
6.6.3	Special Derived Functions . . . . .	36
6.6.4	An Example . . . . .	37

## 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

CoreASM engine with Carma can be downloaded from the web at [www.coreasm.org/download](http://www.coreasm.org/download).

#### 1.1.1 System Requirements

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

#### 1.1.2 Installing Carma

To install CoreASM with Carma just unzip the contents of the binary package into a directory of your choice. Alternatively, You can build CoreASM with Carma using the ant build file 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 from 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

You need to have Eclipse 3.5 or newer installed on your machine.

#### 1.2.2 Installing CoreASM Eclipse Plugin

To install the latest development version of the CoreASM Eclipse plugin using the update site, run Eclipse and follow these steps:

1. Uninstall and remove any previously installed CoreASM Eclipse plugin version 0.4.x or older.
2. Under *Help* → *Install New Software...* add an update site with the following information:  
*Name:* CoreASM Eclipse Plugin trunk  
*URL:* <http://www.coreasm.org/eclipse/update-trunk>
3. Select the “CoreASM Eclipse Plugin Features” category and continue with the installation.

The development version of the CoreASM Eclipse plugin is tested on Eclipse 3.5.

### 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* → *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. 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.

## 2 CoreASM Specification

Figure 1 shows a typical structure of a CoreASM specification<sup>1</sup>. Every specification starts with the keyword **CoreASM** followed by the name of the specification. Plugins that are required in the specification are then listed one by one with the keyword **use** followed by the name of the plugin.

<sup>1</sup> 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.

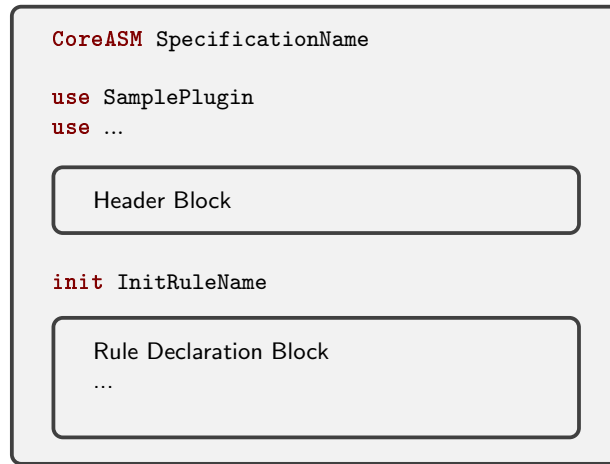


Figure 1: Typical Structure of a CoreASM Specification

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](#).

## 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](http://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 Spec 2.1, 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
    par
      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 to the agent that is running the enclosing rule.

CoreASM kernel also defines a couple of important operators:

▷ `value1 = value2` ..... Kernel



This is the equality operator.

▷ **ruleelement** *id* ..... Kernel

This operator returns the rule element of a rule with the given name (*id*).<sup>1</sup> 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.

▷ **@** *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:

► **loc** := *value* ..... 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.pluginLoadRequest** a comma separated list of plugins to be loaded in addition to those listed in the specification being loaded.

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

This plugin provides the following rule form:

Chooses an element from the *enumerable*<sup>2</sup> *value* which satisfies *guard*, assigns it as the value of *id* and evaluates *rule*<sub>1</sub>. If the **ifnone** clause is provided, *rule*<sub>2</sub> will be evaluated if no element can be found. The last keyword **endchoose** is optional.

See Section 4.7 for more information on **forall** expressions.

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

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

<sup>2</sup>An *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 is the not-equal operator which is defined on all elements. The semantics of this operator is equivalent to "**not** ( $value_1 = value_2$ )".

▷  $value_1$  bin-op  $value_2$  ..... 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.

▷  $value_1$  bin-op  $value_2$  ..... Number Plugin

Performs binary operations on number values. Currently supported operators are

$+$   $-$   $*$   $/$  **div** **%**  $>$   $\geq$   $<$   $\leq$

which result in Number or Boolean values.

▷  $|value|$  ..... Number Plugin

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

◆ **infinity**:  $\rightarrow$  NUMBER ..... Number Plugin

Is the positive infinity.

◆ **toNumber**: ELEMENT  $\rightarrow$  NUMBER ..... Number Plugin



```

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.

▷ *id* (*value*<sub>1</sub>, ..., *value*<sub>*n*</sub>) (*value*'<sub>1</sub>, ..., *value*'<sub>*m*</sub>) ..... Kernel Extensions Plugin

▷ (*value*) (*value*'<sub>1</sub>, ..., *value*'<sub>*m*</sub>) ..... Kernel Extensions Plugin

The above two forms apply the arguments *value*'<sub>*i*</sub> to the function element at location (*value*) (*value*<sub>1</sub>, ..., *value*<sub>*n*</sub>) 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 **bar** are two defined functions, and *bar* = @foo:

```

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

```

► **call** *id* (*value*<sub>1</sub>, ..., *value*<sub>*n*</sub>) (*value*'<sub>1</sub>, ..., *value*'<sub>*m*</sub>) ..... Kernel Extensions Plugin



► **call** (*value*) (*value*'<sub>1</sub>, ..., *value*'<sub>*m*</sub>) ..... Kernel Extensions Plugin

The above two rules call the rule element value of *id* (*value*<sub>1</sub>,..., *value*<sub>*n*</sub>) (the first form) or *value* (the second form) with the arguments *value*'<sub>*i*</sub>. For example, if we have `foo(5) = @MyRule` and

```
| rule MyRule (a,b) =
|   print a + " talks to " + b
```

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.

► **abstract** *value* ..... Abstraction Plugin

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



the generated updates in a virtual state, and so on. The resulting update set is a sequential composition of the updates generated by  $rule_1 \dots rule_n$ .

► **iterate** *rule* ..... TurboASM Plugin

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).

► **while** (*value*) *rule* ..... TurboASM Plugin

This rule is equivalent to:

```
iterate
  if value then rule
```

► *loc* ← *rule* ..... TurboASM Plugin

Replaces all the occurrences of **result** in *rule* with *loc* and evaluates the rule. 1 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)
```

◆ **return** *value in rule* ..... TurboASM Plugin

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.

► **local**  $\frac{id_1, id_2, \dots, id_n}{optional}$  **in** *rule* ..... TurboASM Plugin

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 a small set of functions and expression forms.

▷  $value_1 + value_2$  ..... 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."
```

◆  $toString: ELEMENT \rightarrow STRING$  ..... String Plugin

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

◆  $strlen: STRING \rightarrow NUMBER$  ..... String Plugin

Returns the length of the given String value.

◆  $matches: STRING \rightarrow STRING$  ..... 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:

► **print** *value* ..... IO Plugin

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

◆ *input*: **STRING** → **STRING** ..... IO Plugin

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 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
  else
    program(self) := undef
  end

```

## 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.

◆  $foldl : \text{ELEMENT} \times \text{FUNCTION} \times \text{ELEMENT} \rightarrow \text{ELEMENT}$  ..... Collection 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.

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

◆  $foldr : \text{ELEMENT} \times \text{FUNCTION} \times \text{ELEMENT} \rightarrow \text{ELEMENT}$  ..... Collection Plugin

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

$$\text{foldr}([x_1, \dots, x_n], f, i) \equiv f(x_1, f(x_2, \dots f(x_n, \text{init}))) \dots$$

◆  $\text{fold} : \text{ELEMENT} \times \text{FUNCTION} \times \text{ELEMENT} \rightarrow \text{ELEMENT}$  ..... Collection Plugin

This is the same as  $\text{foldr}$ ; see above.

◆  $\text{map} : \text{ELEMENT} \times \text{FUNCTION} \times \text{ELEMENT} \rightarrow \text{ELEMENT}$  ..... Collection Plugin

$\text{map}(\mathbf{c}, \text{@func})$  applies the unary function  $\text{func}$  to all the elements of  $\mathbf{c}$  (any collection, such as list and set) and returns a new collection (with the same structure as that of  $\mathbf{c}$ ).

$$\text{map}([x_1, \dots, x_n], f) \equiv [f(x_1), f(x_2), \dots f(x_n)]$$

◆  $\text{filter} : \text{ELEMENT} \times \text{FUNCTION} \times \text{ELEMENT} \rightarrow \text{ELEMENT}$  ..... Collection Plugin

$\text{filter}(\mathbf{c}, \text{@func})$  applies the boolean unary function  $\text{func}$  to all the elements of  $\mathbf{c}$  and returns a new collection with only those elements of  $\mathbf{c}$  for which  $\text{func}$  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.

▷  $\{ \underbrace{\text{value}_1, \dots, \text{value}_n}_{\text{optional}} \}$  ..... Set Plugin

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.

▷  $\{ \text{id} \mid \text{id in value with guard} \}$  ..... Set Plugin  
optional

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

▷  $\{ \text{id is exp} \mid \text{id}_1 \text{ in } \underbrace{\text{value}_1, \dots, \text{value}_n}_{\text{optional}} \text{ with } \underbrace{\text{guard}}_{\text{optional}} \}$  ..... Set Plugin

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  $id_1, \dots, id_n$  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}

▷ *value<sub>1</sub> bin-op value<sub>2</sub>* ..... Set Plugin

Performs a set binary operation where both *value<sub>1</sub>* and *value<sub>2</sub>* 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*.

► **remove value from 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 removes *value* to *loc*.

## 5.9 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.

▷ [ *value<sub>1</sub>, value<sub>2</sub>, ..., value<sub>n</sub>* ] ..... List Plugin  
                                   *optional*



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

▷  $value_1 + value_2$  ..... List Plugin

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

◆  $toList: ELEMENT \rightarrow LIST$  ..... List Plugin

If  $e$  is an enumerable (e.g., number range, set, etc.),  $toList(e)$  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.

◆  $flattenList: LIST \rightarrow LIST$  ..... List Plugin

If  $l$  is a netsting list,  $flattenList(l)$  will return a flatten version of  $l$ .

◆  $head: LIST \rightarrow ELEMENT$  ..... List Plugin

Returns the first element of the list.

◆  $last: LIST \rightarrow ELEMENT$  ..... List Plugin

Returns the last element of the list.

◆  $tail: LIST \rightarrow LIST$  ..... List Plugin

Returns all but the first element of the list.

◆  $cons: ELEMENT \times LIST \rightarrow LIST$  ..... List Plugin

Creates a new list with the given element as its head and given list as its tail.

◆  $nth: LIST \times NUMBER \rightarrow ELEMENT$  ..... List Plugin

Returns the  $n^{th}$  element of the list. The index of the first element is 1.

◆  $setnth: LIST \times NUMBER \times ELEMENT \rightarrow LIST$  ..... List Plugin

$setnth(list, i, e)$ , if  $i$  is a valid index for  $list$ , returns a new list in which the element at index  $i$  is  $e$ .

◆  $take: LIST \times NUMBER \rightarrow LIST$  ..... List Plugin

<sup>3</sup>The old form of  $\langle\langle x1, \dots, xn \rangle\rangle$  still works but it is deprecated and may not be supported in future releases of the CoreASM engine.

*take*(list, i) returns the first i elements of list *list*.

◆ *drop*: LIST  $\times$  NUMBER  $\rightarrow$  LIST ..... List Plugin

*drop*(list, i) returns what is left after dropping the first *i* elements of the list *list*.

◆ *reverse*: LIST  $\rightarrow$  LIST ..... List Plugin

Returns a list consisting of the given list's elements in reverse order.

◆ *indexes*: LIST  $\times$  ELEMENT  $\rightarrow$  LIST ..... List Plugin

Returns a potentially empty list of the indexes of the given element in given list.

◆ *indices*: LIST  $\times$  ELEMENT  $\rightarrow$  LIST ..... List Plugin

The same as *indexes*; see above.

◆ *zip*: LIST  $\times$  LIST  $\rightarrow$  LIST ..... List Plugin

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.

◆ *zipwith*: LIST  $\times$  LIST  $\times$  FUNCTION  $\rightarrow$  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, @Sum) is applied to two lists to produce a list of corresponding maximums, where **derived** Sum(*x*, *y*) = *x* + *y*.

◆ *replicate*: ELEMENT  $\times$  NUMBER  $\rightarrow$  LIST ..... 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 (shifting 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
    endpar
  endpar
```

## 5.10 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:

► **enqueue value into loc** ..... Queue Plugin

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.

► **dequeue loc<sub>v</sub> from loc<sub>q</sub>** ..... Queue Plugin



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** → **MAP** ..... Map Plugin

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  $\forall [k_i, v_i] \nexists [k_j, v_j] \ k_i = k_j \wedge v_i \neq v_j$ , *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 → "John", 2 → "Mary"}
```

◆ *mapToPairs*: **MAP** → **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 → "John", 2 → "Mary"})

  results in

{[1, "John"], [2, "Mary"]}
```

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

► **add value to loc** ..... Map Plugin

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.

► **remove value from loc** ..... Map Plugin

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.

### 5.13 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):

$$\text{universe } id = \frac{\{ id_1, \dots, id_n \}}{\text{optional}}$$

- Definition of enumeration backgrounds through the following syntax:

$$\text{enum } id = \frac{\{ id_1, \dots, id_n \}}{\text{optional}}$$

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:

$$\text{function } id_f : \frac{id_{u1} * \dots * id_{un} \rightarrow id_r}{\text{optional}}$$

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:

$$\text{derived } id_f \left( \frac{id_1, \dots, id_n}{optional} \right) = expression$$

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

```
| derived f(x, y) = x2 + y2
```

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.

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 i to set values of engine properties:

Other plugins (such as the Signature Plugin, see Section 5.13) 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:

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.

◆ **stepcount** :→ NUMBER ..... Time Plugin

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:

► **debuginfo** *id value* ..... DebugInfo Plugin

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 **DebugInfo.activeChannels** engine property. This can be done either through the Options plugin or by setting the values directly from the engine driver (e.g., **Carma**). For example, using the Options plugin one can add the following line to a spec to turn the logging on for channels *warning* and *error*:

```
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 DebugInfo.activeChannels ALL
//option DebugInfo.activeChannels "ch1 ch2"
//option DebugInfo.activeChannels "ch1,ch2"
//option DebugInfo.activeChannels ch1
//option DebugInfo.activeChannels NONE

init R1

rule R1 = {
  if mode = undef then {
    debuginfo ch1 "initializing."
    mode := "counting"
    counter := 0
  } else {
    debuginfo ch2 "counting."
    counter := counter + 1
  }
}
```

## 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  $\pi$ , 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  $\pi$ .
- **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  $(r, \theta)$  and returns  $\theta$ .
- **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.
- **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.
- **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*.
- **powerset({e1,...,en})** This function returns the powerset of the given set of elements.

### 6.6.4 An Example

```

CoreASM MathPluginExample

use StandardPlugins
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
    printprint "log(e) = " + log(e)
    print "sin(30) = " + round( sin( toRadians(30) ) * 10 ) / 10
    print "asin(0.5) = " + round( toDegrees( 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)
    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})"
      else
        print x + " is not a member of powerset({1, 2, 3})"
    endpar
  endpar

```

**CoreASM Spec 6.1:** Using Math Plugin

As an example, the output of the CoreASM Spec 6.1 would be the following:

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