How to Write a CoreASM Plug-in

First Draft: February 13, 2014

www.coreasm.org

Roozbeh Farahbod
rfarahbo@cs.sfu.ca

Copyright © 2007
Your criticism is welcome.

Contents

Plug-in Interface	Extends	Description
Parser Plug-in	Parser	provides additional grammar rules to the parser
Interpreter Plug-in	Interpreter	provides new semantics to the interpreter
Operator Provider	Parser, Interpreter	provides grammar rules for new operators along with their precedence levels and semantics
Vocabulary Extender	Abstract Storage	extends the state with additional functions, universes, and backgrounds
Aggregator	Abstract Storage	aggregates partial updates into basic updates
Scheduler Plugin	Scheduler	provides new scheduling policies for multi-agent ASMs
Extension Point Plugin	all components	extends the control state model of the engine

Table 1: CoreASM Plug-in Interfaces

1 The CoreASM Plug-in Framework

A CoreASM plug-in is a Java class derived from the abstract class `Plugin`[1] that typically also implements one or more of the interfaces defined by the CoreASM extensibility framework (see Table 1). The framework supports two extension mechanisms: plug-ins can either extend the functionality of specific components of the engine, by contributing additional data or behavior to those components, or they can extend the control state ASM of the engine itself, by interposing their own code in between state transitions.

1.1 Parser Extensions

Plug-ins can implement the `ParserPlugin` interface and/or the `OperatorProvider` interface to extend the parser by respectively contributing additional grammar rules and/or new operator descriptions. Before parsing a specification, the engine gathers all the grammar rules and operator descriptions provided by all parser plug-ins and operator providers. The parser component then combines these grammar rules and operator tokens with the kernel grammar and builds a new 'parser' to scan the specification. While building the abstract syntax tree, this parser labels the nodes that are created by plug-in-provided grammar rules with the plug-in identifier; these labels can later be used by the interpreter to evaluate such nodes.

[1] `org.coreasm.engine.plugin.Plugin`

1.2 Interpreter Extensions

Plug-ins can extend the interpreter of the engine by implementing either the `InterpreterPlugin` interface or the `OperatorProvider` interface (or both). These plug-ins provide the semantics for grammar rules and operations contributed by the plug-in.

As already mentioned earlier, nodes of the parse tree corresponding to grammar rules provided by a plug-in are annotated with the plug-in identifier. If a node is found to refer to a plug-in, the interpreter calls the plug-in to obtain the semantics of the node; otherwise, the default kernel interpreter is used to evaluate the node. A similar approach is also used by the engine to obtain semantics of extended operators from Operator Providers.

1.3 Abstract Storage Extensions

Plug-ins that implement the `VocabularyExtender` interface can extend the vocabulary of the CoreASM state by contributing new backgrounds, universes, and functions to the abstract storage. Such plug-ins in fact extend the initial state and signature of the simulated machine.

Plug-ins can also implement the `Aggregator` interface and provide aggregation rules to be applied on update instructions before they are submitted to the state. The aggregator plug-ins are called to aggregate update instructions before the engine applies the updates to the state. Aggregators are used, for example, to implement partial updates such as *add* and *remove* operations on sets.

1.4 Scheduler Extensions

CoreASM plug-ins can implement `SchedulerPlugin` to extend the scheduler of the engine by providing new scheduling policies that affect the selection of agents in multi-agent ASMs. They provide an extension to the scheduler that is used to determine at each step the next set of agents to execute. It is worthwhile to note that only a single scheduling policy can be in force at any given time, whereas an arbitrary number of plug-ins of the remaining types can be all in use at the same time.

1.5 Extension Point Plug-ins

In addition to modular extensions of specific components, plug-ins can also extend the control state of the engine by implementing the `ExtensionPointPlugin`. Each mode transition in the execution engine is associated to an extension point. At any extension point, if there is any plug-in registered for that point, the code contributed by the plug-in for that transition is executed before the engine proceeds into the new mode. Such a mechanism enables arbitrary extensions to the engine's life-cycle, which facilitates implementing various practically relevant features such as adding debugging support, adding a C-like preprocessor, or performing statistical analysis of the behavior of the simulated machine (e.g., coverage analysis or profiling).

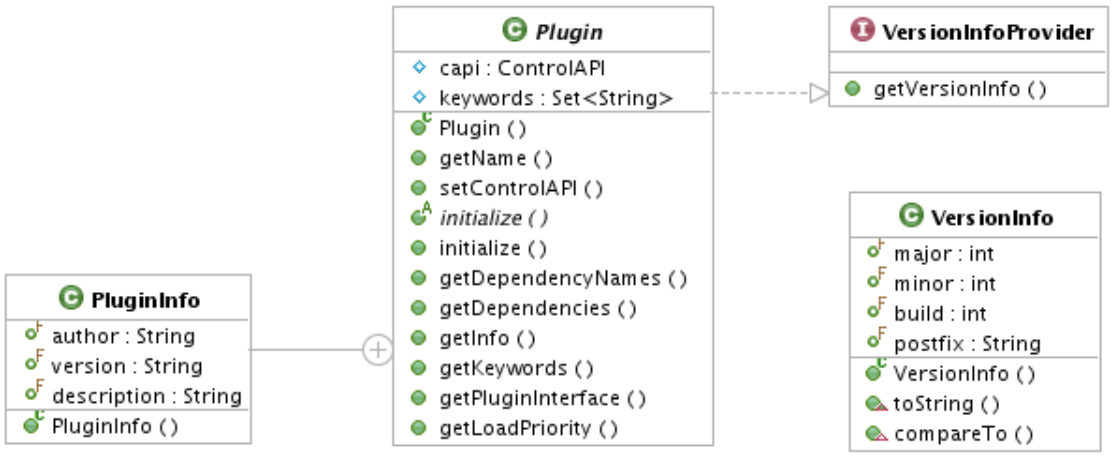

Figure 1: CoreASM `Plugin` Abstract Class

2 Creating A Basic Plug-in

In this section we create a basic CoreASM plug-in and load it into the engine. We start by looking into the `Plugin` class. We then create our sample plug-in class and deploy it to the engine.

2.1 The `Plugin` Class

The `Plugin` class provides the core structure of CoreASM plug-ins (see Figure 1). In its basic form, every CoreASM plug-in should have a name, an initialize method, and a set of dependency requirements.

- `getName()` returns the name of this Plug-in, which is the name of the runtime class of this plugin.
- The abstract method `initialize()` is called by the engine to initialize the plug-in. CoreASM plug-ins implement this method to perform custom initialization; e.g., registration of operators in the engine.
- `getDependencyNames()` provides the names of the plugins that this plugin depends on. It should return an empty set if there is no dependency requirement for this plugin. By default, it returns an empty set.
- `getDependencies()` provides a map of *name* → *version-information* of the plugins on which this plugin depends on. The version-information is the minimum version the required plugin should have. This method should return an empty map if there is no dependency requirement for this plugin. By default, this method will return the dependency names (see above) with a minimum version of '0.0.0'.

- `getInfo()` returns some information about this plugin in an instance of `PluginInfo` object.
- CoreASM plug-ins can override `getPluginInterface()` to return an interface to provide custom services to engine's environment (i.e., GUIs, tools, etc.).
- `getLoadPriority()` returns the suggested loading priority of this plug-in. Zero is the lowest priority and 100 is the highest loading priority. The engine will consider this priority when loading plug-ins. All plug-ins with the same priority value will be loaded in a non-deterministic order.

2.2 Our First Plug-in

As we mentioned earlier, CoreASM plug-ins are subclasses of the abstract class `Plugin`. As a result, every CoreASM plug-in should at least implement the following two methods: `initialize()` and `getVersionInfo()`.

To create our first plug-in, we create a Java class `FirstSamplePlugin.java` (see Program 1) extending the `Plugin` class. In this plug-in, we don't need to do anything for initialization, so we leave the method empty. For version information, we create a new static instance of `VersionInfo` and return it in the `getVersionInfo()` method.

Program 1 `FirstSamplePlugin.java`

```
package myplugins.firstplugin;

import org.coreasm.engine.VersionInfo;
import org.coreasm.engine.plugin.Plugin;

public class FirstSamplePlugin extends Plugin {

    public static final VersionInfo verInfo = new VersionInfo(0, 1, 1, "alpha");

    @Override
    public void initialize() {
        // do nothing
    }

    public VersionInfo getVersionInfo() {
        return verInfo;
    }

}
```

As you can see, our first plug-in basically does nothing! We keep it simple for now and will later improve this plug-in to actually extend various aspects of the engine.

Before we continue, let's organize things a bit. We create a folder for our plugin, with two subfolders `src` and `bin` to keep the source file and the binary file of plug-in.

```
$ cd firstPlugin
$ ls
bin  src
$ tree
.
|-- bin
`-- src
    `-- myplugins
        `-- firstplugin
            `-- FirstSamplePlugin.java
$ _
```

2.3 Deploying Our First Plugin

Now that we created our plug-in class, we need to pack it properly so that the CoreASM engine can load it. This process has four steps:

1. compiling the Java class(es)
2. creating the plug-in id file; every CoreASM plug-in needs to have an identification file that holds the absolute name of the plug-in class
3. packing the compiled class(es) together with the id file into a single JAR file
4. copying the JAR file to the CoreASM plugins folder

Compiling There are many ways to compile your Java class files. Here we simply call the Java compiler from the command line. To compile a plug-in, the binary class files of the CoreASM engine must be in your classpath. To make it short, we assume that these binary files are in '`../CoreASMBin`'.

```
$ javac -cp ../CoreASMBin:bin -sourcepath src -d bin src/myplugins/firstplugin/FirstSamplePlugin.java
$ tree
.
|-- bin
|   `-- myplugins
|       `-- firstplugin
|           `-- FirstSamplePlugin.class
`-- src
    `-- myplugins
        `-- firstplugin
            `-- FirstSamplePlugin.java
$ _
```

Plug-in Id To create the plug-in identification file use your favorite editor to create a plain text file named `CoreASMPlugin.id` with the following line:

```
myplugins.firstplugin.FirstSamplePlugin
```

Move `CoreASMPlugin.id` file to the `bin` folder.

Packaging Now that we have all the required files, we need to pack them together in a JAR file named '`FirstSamplePlugin.jar`'. The JAR file should contain the compiled Java classes (contents of the `bin` folder) together with the identification file.

```
$ cd bin
$ ls
CoreASMPlugin.id  myplugins
$ jar cf ../FirstSamplePlugin.jar CoreASMPlugin.id myplugins/
$ cd ..
$ ls
bin  FirstSamplePlugin.jar  src
$ _
```

Installing The last step is to copy the JAR file to the '`plugins`' folder of the CoreASM engine installed on your system. To check if everything is done properly, we can run Carma with '`--version`' option to get a list of all the installed plug-ins:

```
$ carma --version
Carma 0.5.0-beta by Roozbeh Farahbod
CoreASM Engine 0.9.1-beta
Plugins:
   PlotterPlugin 0.2.0-alpha
   NumberPlugin 0.5.0-beta
   ConditionalRulePlugin 0.9.0-beta
   FirstSamplePlugin 0.1.1-alpha              <----
   Kernel 0.7.1-beta
   TabBlocksPlugin 0.1.0-alpha
   BlockRulePlugin 0.9.0-beta
   ExtendRulePlugin 0.8.0-beta
   TurboASMPlugin 0.8.1-beta
   ChooseRulePlugin 0.9.0-beta
   ForallRulePlugin 0.9.0-beta
   StringPlugin 0.3.0-beta
   IOPlugin 0.3.0-beta
   PredicateLogicPlugin 0.4.1-beta
   LetRulePlugin 0.9.0-beta
   ListPlugin 0.2.0-alpha
```

Congratulations! In the following sections, we focus on improving the plug-in by implementing the interfaces listed in Table 1. We will not repeat the above procedure as it would be virtually the same.

3 Adding A New Function to CoreASM

In this section, we will extend the abstract storage of the engine by adding a new function to the state. To do so, we first look at the structure of a CoreASM state.

3.1 CoreASM State Structure

A CoreASM state is composed of *universes*, *functions*, and *rules*. Values in a CoreASM state are represented by *elements*. Any value in the state is an instance of either the root class `Element` or one of its subclasses. Universes, functions, and rules are also elements. This would allow universes, functions, and rules to be treated as values if needed. Universes are defined by their characterisitc function over elements of the states.

Figure 2 presents a class diagram of the core classes and interfaces contributing to the structure of a CoreASM state.[2] The abstract class `FunctionElement` is the super class of all function elements in the state. Standard dynamic functions in ASM are implemented by the `MapFunctionElement` class which is an implementation of `FunctionElement` using maps. Universes are implemented by the `UniverseElement` class which is itself a sublcass of `FunctionElement`. The `BackgroundElement` class implements backgrounds as static universes.

The interface to a CoreASM state is defined by the Java interface `State` (see Figure 2). Through this interface, other components of the engine can add new functions, universes, and rules or query about existence of those elements. Plug-ins, as well as other components of the engine, can also set and get values of locations of the state using the `getValue` and `setValue` methods of this interface.

3.2 Our Second Plug-in

For our second plug-in, we extend the CoreASM state by introducing a new derived function to calculate a hash value for String elements. We define the signature of this function as:

$$hashValue : \textsc{String} \rightarrow \textsc{Number}$$

To provide the semantics of this function, we extend the `FunctionElement` class by a new class called `HashValueFunction` and we override the `getValue` of this class to calculate a hash value for the input String argument. See Program 2 for the actual implementation of this class. We will see later how our second plug-in instanciates this function and offers it to the engine to be added to the CoreASM state.

[2] Java package: `org.coreasm.engine.absstorage`

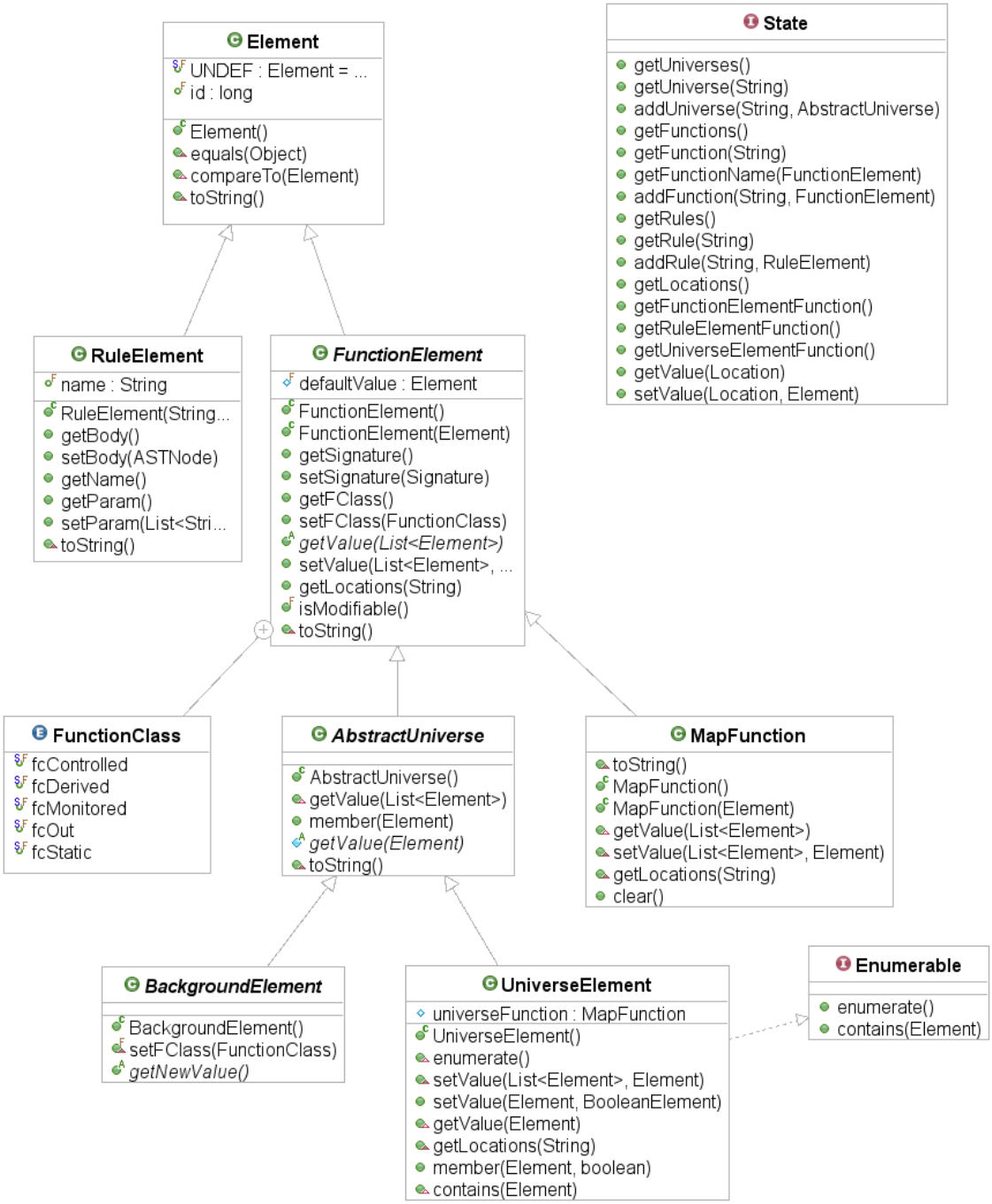

Figure 2: Elements, Rules, Functions, and Universes in Abstract Storage

Program 2 Hash Function Class

```
package myplugins.secondplugin;

import java.util.List;

import org.coreasm.engine.absstorage.Element;
import org.coreasm.engine.absstorage.FunctionElement;
import org.coreasm.engine.stdplugins.number.NumberElement;
import org.coreasm.engine.stdplugins.string.StringElement;

/**
 * A CoreASM function to calculate a hash on String values
 */
public class HashValueFunction extends FunctionElement {

    @Override
    public Element getValue(List<Element> args) {
        Element result = Element.UNDEF;

        // if there is only one argument passed to this function
        if (args.size() == 1) {
            Element argument = args.get(0);

            // if this argument is a StringElement
            if (argument instanceof StringElement) {
                result = new NumberElement(argument.toString().hashCode());
            }

        }

        return result;
    }

}
```

Our second plug-in is a bit more complicated than the first one. We are adding a new function that takes a String element as input and provides a Number element as output. So, our plug-in depends on both the StringPlugin and the NumberPlugin. We should acknowledge this by overriding the `getDependencyNames()` method of the `Plugin` class and instead of an empty set return a set consisting of "StringPlugin" and "NumberPlugin" as Java String objects (see Program 3).

To tell the engine that this plug-in extends the CoreASM state, we implement the `VocabularyExtender` plug-in. This would require us to implement six more methods: `getFunctions()`, `getUniverses`, `getBackgrounds`, and similar ones for the names of these elements. We create a mapping of the String constant "hashValue" to an instance of our `HashFunctionElement` and return this map in the `getFunctions()` method. To keep a single instance of this hash function, we use the `initialize` method to create and save an instance of this function. That's all! The Java source code of our second plugin is presented in Program 3. We can now complile and deploy this plugin as we did for our first plugin.

With this new plug-in, the following CoreASM specification would be valid:

```
CoreASM TestSecondSamplePlugin

use NumberPlugin
use StringPlugin
use SecondSamplePlugin
use IOPlugin

init Main

rule Main =
   print hashValue("R Farahbod")
```

and running it with Carma would result in the following (or similar) output:

```
$ carma -s 1 TestPlugin.coreasm
* Carma * : Loading the specification.
* Carma * : Performing the initial step.
4.9477543E7
* Carma * : Starting the execution.

4.9477543E7

* Carma * : Stopped after 1 steps.
* Carma * : Execution concluded.
$ _
```

Program 3 SecondSamplePlugin.java

```
package myplugins.secondplugin;

import org.coreasm.engine.VersionInfo;
import org.coreasm.engine.plugin.Plugin;
import org.coreasm.engine.plugin.VocabularyExtender;
import org.coreasm.engine.absstorage.*;
import java.util.*;

public class SecondSamplePlugin extends Plugin implements VocabularyExtender {

    public static final VersionInfo verInfo = new VersionInfo(0, 2, 1, "alpha");
    private Map<String,FunctionElement> functions = null;
    private FunctionElement hashValueFunction;
    private final Set<String> dependencySet;

    public SecondSamplePlugin() {
        super();
        dependencySet = new HashSet<String>();
        dependencySet.add("StringPlugin");
        dependencySet.add("NumberPlugin");
    }

    @Override
    public void initialize() { hashValueFunction = new HashValueFunction(); }

    public VersionInfo getVersionInfo() { return verInfo; }

    @Override
    public Set<String> getDependencyNames() { return this.dependencySet; }

    public Map<String,FunctionElement> getFunctions() {
        if (functions == null) {
            functions = new HashMap<String,FunctionElement>();
            functions.put("hashValue", hashValueFunction);
        }
        return functions;
    }

    public Set<String> getFunctionNames() { return functions.keySet(); }

    public Map<String,UniverseElement> getUniverses() { return Collections.emptyMap(); }

    public Set<String> getUniverseNames() { return Collections.emptySet(); }

    public Map<String,BackgroundElement> getBackgrounds() { return Collections.emptyMap(); }
}
```