An Overview of Epsilon *Tools*

Dimitrios S. Kolovos and Louis M. Rose
Department of Computer Science
The University of York

Introduction

Model management languages such as those provided by Epsilon are by design not general-purpose languages. Therefore, there are features that such languages do not support inherently (mainly because such features are typically not always needed in the context of model management).

However, there are cases where a feature that is not built-in may be necessary for a specific task. As an example, consider the need to check for string similarity using advanced string matching algorithms during model comparison. While it is possible to implement such algorithms using an imperative language such as EOL, it is not really advisable since performance-wise the implementation in EOL will be at least one order of magnitude worse than, e.g. Java. Moreover, many such algorithms have already been implemented and are freely available and translating them into EOL would probably be misplaced effort.

In a more elaborate scenario, consider the need to access information stored in a database server during a model management operation. For instance, in the context of acceptance testing we may want to check that for each table in our live database, there is a class with the same name in our UML model that describes the system. Obviously, it is impossible to implement something like JDBC using a task-specific language such as EOL.

Epsilon Tools

To address such issues and enable developers to implement non-standard functionality, Epsilon supports the *Tool*[1] concept. A *tool* is a normal Java class that (optionally) conforms to a specific interface (org.epsilon.eol.tools.ITool[2]) and which can be instantiated and accessed from the context of an EOL (or any other EOL-based language such as EML, ETL, EVL etc) program. After instantiation, EOL can be used to invoke methods and access properties of the object. In this document we show how to implement a new tool and use it from an EOL program.

[1] The concept of *tool* was originally inspired by a similar extensibility mechanism provided by the Velocity (http://jakarta.apache.org/velocity) template engine
[2] Or extends its abstract subclass org.epsilon.eol.tools.AbstractTool

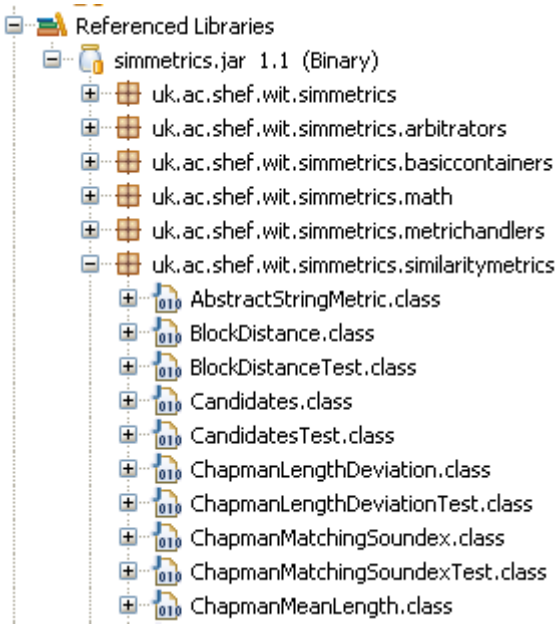
Implementing the SimMetrics Tool

In the context of this example we present a tool called SimMetricsTool. The purpose of this tool is to provide access from EOL to Simmetrics[3], a Java library that contains implementations of several string similarity calculation algorithms. The source code of the SimMetricsTool follows:

```
package org.epsilon.ecl.tools.textcomparison.simmetrics;

import org.epsilon.eol.exceptions.EolInternalException;
import org.epsilon.eol.exceptions.EolRuntimeException;

import uk.ac.shef.wit.simmetrics.similaritymetrics.AbstractStringMetric;

public class SimMetricsTool {

	public double similarity(String s1, String s2, String algorithm)
		throws EolRuntimeException {
		AbstractStringMetric metric;

		if (s1 == null || s2 == null) {
			if (s1 == null && s2 == null ) {
				return 1.0;
			}
			else {
				return 0.0;
			}
		}

		try {
			metric = (AbstractStringMetric) Class.forName(
				"uk.ac.shef.wit.simmetrics.similaritymetrics."
					+ algorithm).newInstance();

			return metric.getSimilarity(s1, s2);
		} catch (Exception e) {
			throw new EolInternalException(e);
		}

	}

}
```

The tool defines a similarity() method which defines three arguments, the strings and the name of the algorithm that is used to compare them, and returns a double which shows the similarity of the two strings according to the algorithm. With respect to the body of the method, after some checking for nulls, the operation attempts to create an instance of the class that implements the specific algorithm dynamically (using Class.forName). For example if the algorithm parameter is set to "ChapmanMeanLength" the method will try to instantiate the uk.ac.shef.wit.simmetrics.similaritymetrics.ChapmanMeanLength class.

[3] Simmetrics is freely available at http://www.dcs.shef.ac.uk/~sam/simmetrics.html



If it succeeds, it invokes its getSimilarity() method, otherwise it fails and throws an exception.

Using the SimMetricsTool from Epsilon

To use the SimMetrics tool from EOL, we need to create an instance of it and then invoke its similarity() method. An example follows:

```
var simmetrics : new Native(
      'org.epsilon.ecl.tools.textcomparison.simmetrics.SimMetricsTool');

var algorithm : String := 'JaroWinkler';

simmetrics.similarity('abc', 'abcd', algorithm).println();
simmetrics.similarity('abc', 'xyz', algorithm).println();
```

We define a variable named simmetrics, the type of which is Native. Then we define the algorithm we want to use (JaroWinkler) and finally we invoke the similarity method for two different pairs of strings. When we run this example, the first call returns 0.94 which shows that the ‘abc’ is very similar to ‘abcd’ while the second returns 0.0 which indicates that ‘abc’ is not at all similar to ‘xyz’.

Enabling Epsilon to Discover User-Defined Tools

In the previous sections we have shown what the implementation of a tool looks like and how it can be used from within Epsilon. In this section we provide the details of the configuration that will allow Epsilon to discover and use a user-defined tool.

Running Epsilon outside Eclipse

To use any user defined tool when running Epsilon outside Eclipse, the only thing that we need to do is to place the tool and its dependencies in the class-path.

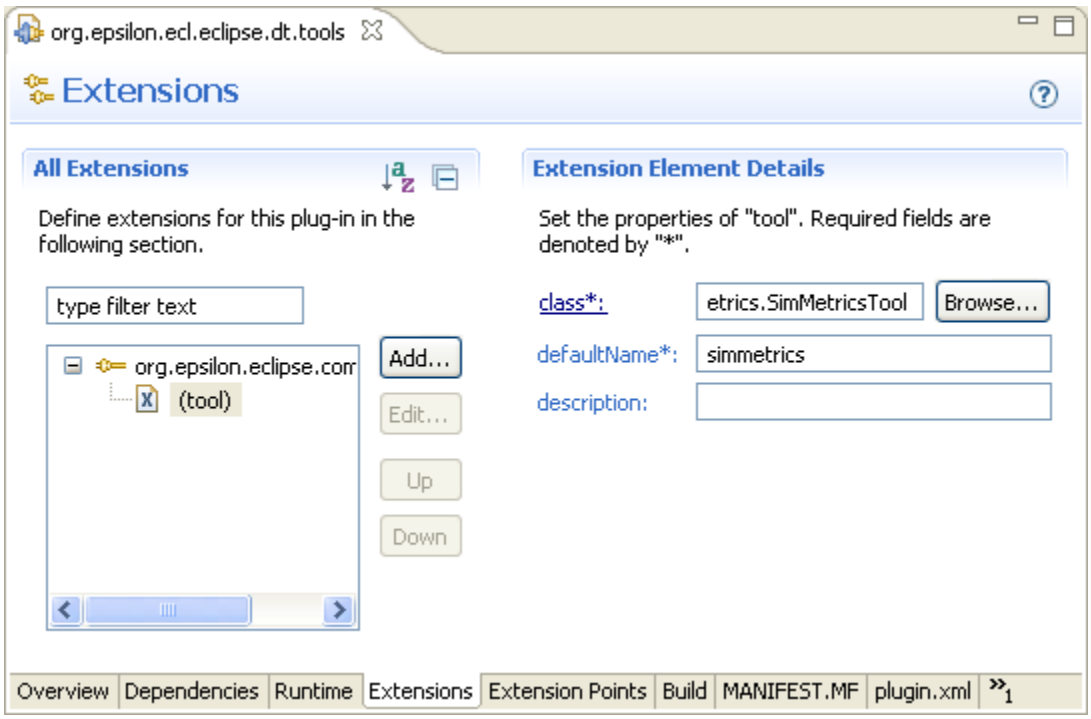
Running Epsilon within Eclipse

To discover user-defined tools within Eclipse, Epsilon defines the org.epsilon.eclipse.common.dt.tool extension point in the org.epsilon.eclipse.common.dt plugin.

Firstly, the project containing your tool must be a plug-in project. (Convert a Java project to a plug-in project by right-clicking the project and clicking PDE Tools | Create plug-in project).

Secondly, the project containing your tool must include org.epsilon.eclipse.common.dt in its dependencies. (Add a dependency by opening a project's META-INF/MANIFEST.MF file and selecting the Dependencies tab. Plug-in dependencies are listed on the left-hand side. Click Add to include an additional dependency).

Finally, the project containing your tool must define an org.epsilon.eclipse.common.dt.tool extension. (Add an extension by opening a project's META-INF/MANIFEST.MF file and selecting the Extensions tab. Click Add to include an additional extension). The org.epsilon.eclipse.common.dt.tool extension point defines three attributes. The class attribute defines the Java class that implements the tool. The defaultName specifies a default name for the tool and the description attribute provides an overview of the capabilities of the tool. The next screenshot shows the declaration of the extension for the SimMetrics tool we have shown above



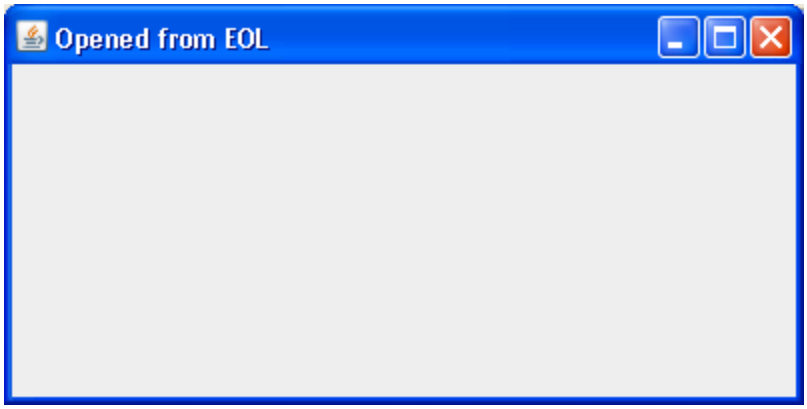
Notes about Epsilon Tools

When we need to implement new tools, the practice we follow is to define the tools in a plugin and define the extensions in another plugin. This keeps the first plugin free of any Eclipse-related dependencies and enables us to reuse it when running Epsilon outside Eclipse.You can find more examples of tools in the org.epsilon.eol.tools plugin[4] and their declaring extensions in the org.epsilon.eol.eclipse.dt.tools plugin[5].

As hinted at above, although it is advisable that your tools implement the ITool interface, so that they can have access to the runtime context (via the getContext() method of ITool), this is not necessary. For instance the following EOL code:

```
var frame : new Native('javax.swing.JFrame');
frame.setBounds(100,100,400,200);
frame.title := 'Opened from EOL';
frame.visible := true;
```

also works fine and when executed opens the following JFrame



[4] http://dev.eclipse.org/viewcvs/index.cgi/org.eclipse.gmt/epsilon/org.epsilon.eol.tools/src/org/epsilon/eol/tools/builtin/?root=Technology_Project

[5] http://dev.eclipse.org/viewcvs/index.cgi/org.eclipse.gmt/epsilon/org.epsilon.eol.eclipse.dt.tools/plugin.xml?revision=1.5&root=Technology_Project&view=markup