Building Modelica Tools using the Modelica SDK

Peter Harman
deltatheta uk ltd.
The Technocentre, Puma Way
Coventry, CV1 2TT, UK
peter.harman@deltatheta.com

Michael Tiller
Emmeskay Inc.
47119 Five Mile Road
Plymouth, MI 48170, USA
mtiller@emmeskay.com

Abstract

Modelica provides numerous opportunities for the engineering industry to promote the reuse and exchange of simulation models by providing a clear standard, open libraries and metadata support via annotations. This opportunity is often underutilized because full Modelica support could not be easily incorporated into software tools without requiring considerable resources.

This paper presents a software development kit, the Modelica SDK, designed specifically to assist developers with integrating Modelica support into any software tool. The philosophy behind this library is to provide maximum extensibility to power users so they can fully utilize the features of the Modelica language and integrate them into their engineering processes for maximum benefit.

The mechanisms provided for a developer to integrate or extend the functionality of the tool into their own software are discussed in detail and examples of the extension points available and their uses are shown.

Keywords: Modelica translator, Java, SDK, API

1 Introduction

This paper presents Modelica SDK, an implementation of Modelica available as a Java library. The Modelica SDK is suitable for adding Modelica support to existing tools as well as developing new Modelica tools and utilities.

Similar tools have been developed with specific goals, such as for style checking and version control [1] as well as translating models for us in an optimization framework [2]. Open source tools [2,3,4] do exist which allow the developer to modify the code to build custom tools. However this not only requires detailed understanding of the underlying software architecture in order to make such modifications but the licensing terms may also be incompatible with the intended purpose of the tool. The aim of the Modelica SDK is to cover all these use cases and allow a broad range of applications, without burdening the developer with creating and maintaining their own implementation of Modelica and/or hindering the developer with undesirable licensing terms.

2 Applications

The aim of this library is to enable the use of Modelica in a wide range of tools and processes and, as such, it is not possible to cover all the possible uses. Instead, we will focus on a number of areas where Modelica could be used in existing tools or where the capabilities of Modelica can be extended using new tools.

2.1 Custom Simulation Platforms

Modelica is first and foremost a modeling language not a simulation platform. Depending on the intended application, the simulation requirements may vary greatly.

For example, in a hardware-in-the-loop (HiL) application it is reasonable to sacrifice a certain degree of accuracy in order to achieve real-time performance as compared to desktop simulation. On the other hand, for detailed design studies, the use of variable time step integrators and careful event handling may dictate greater focus on accuracy at the expense of computational performance. Furthermore, depending on the simulation platform (e.g. HiL hardware, grid computing) I/O requirements may vary (e.g. file system access, IPC support). Finally, domain specific simulation tools, such as engine simulation tools, multi-body tools and/or control system design tools, often have an API for integration of user defined models. In such cases, it may be necessary for the code generation process to support third-party code provided in C, C++, Fortran or some other custom format.

For this reason, the Modelica SDK library exposes a code generation API which allows the developer great flexibility in controlling the code generation process. In this way, the developer can choose to target a wide variety of applications (e.g. Hardware-in-the-loop, desktop simulation), control how I/O and integration of third party code is handled and choose in what particular programming language the target code will be written (e.g. C, assembly, Java).

Some code generation schemes use templates or an intermediate form [5] to provide such flexibility. However these require an extra stage of parsing and also may expose code the developer wishes to remain concealed. For this reason, the Modelica SDK provides a set of Java interfaces allowing developers to implement their own code generator (or subclass the SDK provided ones) to suit their specific needs.

2.2 Parameter Management

It is often desirable to store parameters externally to a model, either for the convenience of editing in spreadsheets or other tools, or simply to comply with company policies requiring the use of standardized formats or a central database [6, 7]. By translating these to and from Modelica records, or providing a method of loading records from this data as described with the path and file system package later, the data can be referenced directly from within Modelica editors by using Modelica dot-notation.

2.3 Parameter Studies and Optimization

Whether using a dedicated optimization package or a spreadsheet, the first step of a parameter study or optimization with a Modelica model is to identify the parameters in the model, their default values and units, and their maximum and minimum values if defined. This can be obtained using the "query" package described later.

Whether a calculated variable depends on a particular parameter is often determined by running multiple simulations in a sensitivity study. However, using the Modelica SDK this information can be determined easily from just the flattened system of equations using API calls (described later in the paper) rather than relying on a brute force numerical approach.

Sensitivities may also be handled during a single simulation with modified ODE solvers which require additional code to be generated by customizing the code generation process using the code generation interfaces.

2.4 Issue Tracking and Version Control

The Modelica SDK also provides the foundation for developing utilities for issue tracking and version control. For example, Modelica aware versions of diff and merge tools could be developed that allow visualization of differences in the model structure rather than just focusing on comparing lines of code. Also, as annotations can be added, queried, modified and removed using the SDK, these capabilities finally make the valuable metadata stored in Modelica accessible for storing additional information about the model, e.g. relating issues (in an issue tracking database) to particular model versions.

2.5 Plug-ins to Vertex

As the Modelica SDK is the Modelica implementation used in the Vertex [8] simulation tool, extensions developed for the Modelica SDK also act as plug-ins to the Vertex tool. In this way, any enhancements or features added through Modelica SDK extension points are therefore also accessible from Vertex. Such extensions might include adding the ability to define new simulation targets, new checking rules or new sources to load Modelica models from (e.g. network servers, version control systems).

3 Interfaces

As a software development kit the ease of integration with different software platforms is essential. Different means of integration have been included for different applications.

3.1 Java

The Modelica SDK is developed in the Java language. This gives advantages of platform-neutrality, convenient packaging as "jar" archive files, and "javadoc" documentation (automatically generated from the Java source code).

The examples given in this paper were all implemented in Java.

3.2 Scripting Languages

All the operations accessible to Java are also accessible by scripting languages running on the Java Virtual Machine (JVM), creating a powerful environment for experimentation. Users of Python (Jython), Ruby (JRuby), Groovy, Beanshell and MATLAB can *directly* access the SDK classes and methods.

3.3 C++ and .NET

The platform neutrality of Java is important for a tool such as the Modelica SDK. However, many applications are developed using other programming languages and platforms. For this reason the key methods in the SDK can be exposed to C++ or .NET applications. Note that in such circumstances the SDK still runs in the Java Virtual Machine, and extensions must still be written in Java.

3.4 Web Services

For complete portability a web services interface has been developed using SOAP. SOAP clients libraries are available in most programming languages. The SOAP services were implemented using Axis2 [9], which comes in both Java and C versions.

4 Capabilities

The SDK provides a Modelica 3.1 compatible parser and object model as well as a translator implementing most features of Modelica 3.1. In this section, we will discuss the various features in the SDK and how they impact both users and developers.

4.1 File and Path Management

For developers, the SDK handles all loading and saving of files. Classes are automatically loaded to memory when required thereby freeing the developer from the tedious tasks of managing memory or specifying which files to load.

4.2 Parsing, Object Model and Manipulation

An important design feature of the SDK is that loaded classes are treated as a set of “live” objects, which can be manipulated and queried in memory (similar to how the DOM object model is handled for XML objects). Finding of a particular component, annotation value or equation, is performed by traversing this object-graph. Every object, whether it represents a Modelica class, component, modification or annotation, can have listeners added to it which are notified of any changes. This makes the SDK ideal for editors and other interactive tools.

Modelica classes represented using the SDK's object model can be transformed into other representations, e.g. written back out in Modelica syntax, using the PrintTarget class. By supplying a custom PrintTarget object, the developer can control the appearance and format of the output, e.g. producing XML, HTML or other formats.

4.3 Querying

A package is included for defining and executing queries on Modelica classes or components. This allows selection of components or classes which match a specified predicate. The Apache Commons Collections [10] library defines an interface called Predicate with one method to evaluate whether an object should be accepted or not. In addition to the interface itself, a number of useful predicates are also provided, such as selecting a component by type or variability, or selecting a class by classes it extends from.

For selecting components there is an additional Boolean flag which controls whether or not to iterate through the hierarchy of the model. If this flag is false, only components at the top-level of the class are returned. If true, components from any level of the class are considered. When selecting classes, a similar additional argument controls whether to restrict queries to only models that are in memory or to iterate over all classes found in the path (thus triggering these classes to be automatically loaded as needed). The latter is, of course, quite slow as all relevant files will be loaded and processed, but it can be useful to consider all classes in the path for some applications.

As an example, the following code selects all the parameters from a specified class:

```
Predicate<ModelicaComponent> predicate =
     new ComponentVariabilityPredicate(
     Variability.PARAMETER );
List<ModelicaComponent> parameters =
     Query.selectComponents(myClass,
     predicate, true);
```

Similarly, the following code selects all loaded classes that extend from Real:

```
Predicate<ModelicaClass> classPredicate
     = new SubclassOfPredicate(“Real”);
List<ModelicaClass> realTypes =
     Query.selectClasses(classPredicate,
     false);
```

4.4 Checking

The “checking” package provides convenient methods for validating a model against the Modelica specification and reporting any issues found. The rules provided check for a variety of different types of issues but the real power in the checking capabilities comes from the ability of developers to *add* their own rules to the system (as discussed later in this paper).

When a developer performs a check on a class, the result is a `CheckResultSet` object which contains the set of all results for rules the class has triggered. Each result has a severity and the set as a whole can be queried as to the highest severity result in the set.

There are two powerful features available once a `CheckResultSet` has been obtained. First, it is possible to add a listener to be notified when a result changes. In this way, the system can automatically update these results as users make changes. Second, each result provides a set of actions (extending `javax.swing.Action`) which can be performed to correct the error. This feature is exploited in Vertex to provide a “one-click” auto-correction mechanism where applicable.

```
CheckResultSet crs =
      CheckManager.checkClass(myClass);
if(Severity.ERROR==crs.getSeverity()) {
      // handle an error
}
```

4.5 Flattening and Symbolic Processing

“Flattening” of a class, from hierarchical form to a flat list of equations and variables, is divided into two steps, “instantiation” and “elaboration”. The instantiation process flattens the hierarchy to a set of variables, equations, connectors and connections, each using dot-notation. The elaboration process performs the symbolic processing.

The symbolic processing handles over-constrained connections [11], generates connection equations, and divides the system by the variability of equations. For each level of variability it then assigns the causality of the equations, and for continuous equations reduces the DAE index by differentiating selected equations.

The following code fetches a class from the Modelica Standard Library, instantiates it, elaborates it and requests the total list of equations:

```
ModelicaClass cc =
    ModelicaClassLoader.findClass(
    “Modelica.Mechanics.Rotational.Examp
    les.CoupledClutches”);
Model model =
    DefaultInstantiator.instantiate(cc);
model.elaborate();
List<Equation> equations =
     model.getEquations();
```

4.6 Code Generation

As mentioned previously, the SDK includes methods to translate a model to code. The flexible code generation architecture is provided through extension points which are discussed in greater detail shortly.

5 Limitations

The design objective for the Modelica SDK is to cover as much of the Modelica specification [12] as possible. The parser and object model cover the full 3.1 specification, so operations which manipulate, query, check or flatten classes, as discussed earlier, operate on Modelica 3.1 code, these are tested on the Modelica Standard Library 3.1. The current limitations are in the symbolic processing and code generation sections of the tool. The current limitations are:

- Overloaded operators are not yet supported
- Expandable connections do not yet generate equations
- Stream connectors do not yet generate equations
- Subtasks are not yet generated and `Subtask.decouple(x)` defaults to `x`
- `semiLinear` is not optimized
- MultiBody extensions are supported but not optimized, e.g. `rooted(x)` defaults to `true`
- Inverse annotations are not yet supported, currently symbolic solutions of equations can be defined as described later
- Reducing systems of equations via methods such as tearing is not currently performed, though this is in development and some other optimizations are currently performed.

6 Extension Points

The SDK provides a number of operations which the user can replace or add to with their own code. These are referred to as extension points. Some of these will be described here. In order to understand how extension points work some knowledge of the Modelica translation process as well as the relationship between Modelica classes and files is required. The extension mechanism makes use of the Java `ServiceLoader` mechanism [13] which eases the development of modular applications. The extension points are defined in the SDK, but extensions can be provided in separate modules on the Java classpath.

6.1 Modelica Path

The Modelica Specification describes the relationship between Modelica classes and files on the file-system. It further describes the operation of the Modelica path lookup mechanism for locating files which uses URLs of the form "modelica://…" to identify the location of files relative to Modelica definitions. The SDK contains interfaces which represent the generalized storage model (for both source files and other types of data contained in the package hierarchy, such as images, data, etc). Implementations of these interfaces are provided for two methods for two such storage schemes. The first is the traditional file system based approach. The second supports accessing code and data from an archive file.

The archive file implementation is based upon a proposal for Modelica 3.1 which allows an entire package (or collection of packages) to be bundled together as a single file. An additional part of that same proposal, the ability to resolve "modelica://" URL's to locate resources such as image or data files from within the archive file itself, is also supported.

This is one of the most powerful features of the SDK because Modelica code, data, images, etc. can all be bundled into a single archive and easily distributed (rather than stored as an elaborate directory structure on the file system). Furthermore, this feature can be used to create an extensible "virtual package hierarchy" where additional resources are mapped to a Modelica package structure and can thus be referenced directly in Modelica code.

An example use of this feature would be the loading of parameter data from a database. By defining two Java classes a database table could be expressed as a package with a series of records contained within it, and these could be referenced within other Modelica code.

6.2 Checking Rules

An important feature of the SDK is the ability to evaluate rules to check a class for validity and find errors. A number of rules are built-in for checking compatibility of a model to the Modelica Specification. By providing an extension of the class `com.deltatheta.modelica.check.CheckRule` additional rules can be added.

A check rule has a single method, to check a class. This method is passed a `CheckResultSet` object, and is expected to add a `CheckResult` instance to the set if the class fails to pass the rule.

```
public class UseOfPartialClassCheck
     implements CheckRule {

  public void checkClass(CheckResultSet
     results, ModelicaClass clazz) {
      for (ModelicaComponent component :
     clazz.getComponents()) {
        try {
          ModelicaClass type =
     component.getModelicaClass();
          if (type.isPartial()) {
               addResult(results,
     component, type, clazz);
          }
      } catch
     (ClassDefinitionNotFoundException
     ex) {           }
    }
  }

  private void addResult(CheckResultSet
     results, final ModelicaComponent
     comp, final ModelicaClass type,
     final ModelicaClass clazz) {
        results.add(new
     AbstractCheckResult() {
        public String getDescription() {
                return ("Component " +
     comp.getName() + " in class" +
     clazz.getPath() + " is of type " +
     type.getPath() + " which is
     partial");
            }
        public Severity getSeverity() {
                return (Severity.ERROR);
        }
      });
    }
}
```

6.3 Symbolic Rules

The SDK has the facility to add rules for solution of scalar equations. This is useful for cases where a particular format of equation is known to appear regularly in a modeling domain but is not solved by the SDK, or where a user defined function has a particular solution. For example, the following code adds the solution of a simple linear equation:

```
Solution.defineRule("&y=&m*&x+&c",
     "&x:=(&y-&c)/&m");
```

6.4 Code Generation

Where the SDK is being used to generate simulation code there is a code generation stage in the translation process. In order to support the diverse range of applications proposed this code generation must be able to be language and platform independent. This is achieved by the code generator itself being an extension matching a specified interface. Note there is no built-in code generator provided in the SDK. Furthermore, the specification of the code generation interface is commercially sensitive and therefore only available to licensed developers.

7 Conclusions

Enabling a product to edit, simulate, import or extract data from Modelica models is no longer a lengthy process. With the availability of the Modelica SDK such capabilities can now be rapidly developed or integrated into existing tools. This dramatically eases the adoption of Modelica as a standard for systems model development and exchange. Furthermore, it creates many opportunities for exploiting currently underutilized features in Modelica (e.g. annotations) and integrating Modelica into the engineering process.

References

[1] Tiller M. *“Parsing and Semantic Analysis of Modelica Code for Non-Simulation Applications”*. Modelica 2003.

[2] Åkesson J., Hedin G., Ekman T., *“Development of a Modelica Compiler using JastAdd”*, Seventh Workshop on Language Descriptions, Tools and Applications. 2007.

[3] Najafi, M., Nikoukhah, R., Steer, S., Furic, S. *“New features and new challenges in modeling and simulation in Scicos”*. 2005 IEEE Conference on Control Applications

[4] Fritzson, P., Aronsson, P., Lundvall, H., Nyström, K., Pop, A., Saldamli, L., Broman, D. *“The OpenModelica Modeling, Simulation, and Development Environment”*. SIMS 2005.

[5] Jonas Larsson and Peter Fritzson. *“A Modelica-based Format for Flexible Modelica Code Generation and Causal Model Transformations”*. In Proceedings of the 5th International Modelica Conference (Modelica'2006), Vienna, Austria, Sept. 4-5, 2006.

[6] Tiller M. *“Implementation of a Generic Data Retrieval API for Modelica”*. Modelica 2005.

[7] Koehler J., Banerjee A. *“Usage of Modelica for transmission simulation at ZF”*. Modelica 2005

[8] deltatheta Vertex. [online] http://www.deltatheta.com/products/vertex/

[9] Apache Software Foundation. Apache Axis2. [online] http://ws.apache.org/axis2/

[10] Apache Software Foundation. Commons Collections. [online] http://commons.apache.org/collections/

[11] Otter M., Elmqvist H., Mattsson SE. *“The New Modelica MultiBody Library”*. Modelica 2003.

[12] Modelica 3.1 Specification. 2009.

[13] O’Conner J. *“Creating Extensible Applications with the Java Platform”*. [online] http://java.sun.com/developer/technicalArticles/javase/extensible/. 2007.