Effective Version Control of Modelica Models

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

Abstract

This contribution introduces Converge, a specialized Version Control System client application designed purely for Modelica. Conventional VCS clients and diff tools cannot inform the user what the effect of a single edit has on the model as a whole. Converge compares selected revisions of a model, loading the Modelica code directly from the VCS repository. This paper presents examples of Modelica code where an edit that appears significant in a conventional diff tool can be shown as not so, and an edit that appears insignificant in a conventional diff tool actually has significant changes to the resulting model.

Successfully comparing two revisions of a model requires resolving the types of components, including handling inheritance, imports and redeclarations. It requires handling of equations and component values, and flattening of the model structure.

Converge includes a complete Modelica implementation, and presents the VCS repository to the user with a number of views, including Packages, Inheritance, Dependencies, Annotations, and Components views; and Instance and Equations views that compare the instantiated model. Changes, and whether they affect the model results, are highlighted to the user. This will allow users to understand the development of models over time and to solve problems caused by changes in dependent Modelica libraries.

Keywords: software configuration management; version control; model lifecycle management

1 Introduction

As a textual modeling language Modelica [1] allows typical Software Configuration Management (SCM) practices to be used, in particular version control. An important part of version control is the "diff" functionality, allowing the user to see changes between revisions.

The goal of this paper is to introduce a new tool, called Converge, for comparing revisions of Modelica models in order to locate sources of errors, determine which changes have potential effects on model results, and track changes of models over time. Converge connects directly to a version control repository to access models.

The paper is structured as follows. Section 2 is an introduction to version control systems and their use with Modelica including the current limitations. Section 3 describes Converge and the range of views it provides of a model. Section 4 gives some examples where Converge gives a more realistic view of the changes to the model than traditional diff tools. Section 5 describes the implementation of the software, and Section 6 concludes.

2 Version Control Systems

Version Control Systems (VCS) store program code along with a database of revisions, and via client applications, allow the user to access a file or set of files for any revision. Usually this revision can be specified as a date or a revision number. Traditional VCS such as Subversion (SVN) [2] or CVS [3] use a central repository. There is also a new breed of Distributed Version Control Systems (DVCS) such as Git [4] or Mercurial [5] where each working copy also has its own complete copy of the repository. The

style of VCS used however does not affect the benefits and limitations of using VCS with Modelica.

The user normally interacts with the VCS with a client application. These can be standalone; integrated into a file explorer such as the popular TortoiseSVN [6] client; or integrated into an Integrated Development Environment such as Visual Studio [7], Eclipse [8] or Netbeans [9]. These generally have either a built-in diff application, or can launch an external diff application with a pair of revisions.

Web based respository browser applications allow viewing of the code without a client application, for example the Trac [10] system used by the Modelica Association [11]. These usually do not contain diff functionality, but do allow viewing of "changesets", which list all files changed and a summary of the changes. These give the user a view of the current state, or the state for a particular revision.

2.1 Diff Tools

The output of a diff tool can be used to interpret the possible effects of changes locally. Within one code file, lines of code inserted, changed or removed can be seen, which for a programming language can be interpreted to see the change in behavior. It doesn't however inform the user, for example, whether dependencies have changed or what the effect of a change of import statement or variable type has on the overall program.

In equation based languages such as Modelica or VHDL-AMS [12] it is rarely even possible to determine locally the change in behavior because the code is not algorithmic.

2.2 Other Limitations of VCS and Modelica

Using version control with a Modelica library is better than not using version control. However there are limitations to the information available to the user from the conventional VCS client.

Loading, editing and saving a Modelica model with a Modelica tool may not preserve exactly the same formatting and whitespace as in the original file.

Annotations have no effect on simulation results; however a large proportion of edits within a Modelica diagram editor will be on the annotations. Each of these edits will be shown in a diff tool and it is currently up to the user to determine which edits are of significance.

Some operations may appear to be significant and yet have no effect on the overall set of variables and equations. Such an example is illustrated later.

3 Viewing Modelica Revisions in Converge

Converge [13] is a standalone tool designed purely for comparing revisions of Modelica models and packages. In essence it is a specialized VCS client designed purely for Modelica. Successfully comparing two models requires resolving the types of components, including handling inheritance, imports and redeclarations. It requires handling of equations and component values, and flattening of the model structure, and therefore has similar needs to a Modelica modeling tool.

The aim of Converge is to successfully solve the problem of version control of Modelica, overcoming the limitations discussed and allowing "model lifecycle management". Ultimately the aim is to answer typical questions that arise during development or utilization of Modelica libraries:

- Does this change affect the results?
- Why does my model give different behavior to two weeks ago?
- What are the dependencies of my model and which have changed?

This is done by providing the user with a range of views, both of the package structure, and of an individual class.

3.1 Global Views

The user defines the path from which Modelica code is loaded. Rather than just a directory or set of directories as in a modeling tool, the path can contain multiple version control repositories. All views are a comparison between 2 revisions, which can be selected. One of these can be set as a local working directory, so the comparison is between a working version and a committed revision, or it can be between 2 revisions.

There are two overall views, one for the path and one for the Modelica packages.

3.1.1 Path View

The Path view shows all files available to be loaded on the Modelica path. It is not discriminated between the source of the files, whether they are stored in directories, in a version control repository or a Modelica archive file, the resulting tree is the same.

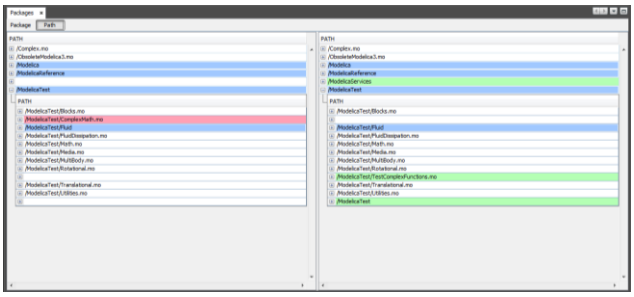

Figure 1: Path View showing revisions of Modelica Standard Library

3.1.2 Packages View

The Packages view shows the hierarchy of Modelica packages as a tree. Similar to the package browser in a Modelica modeling tool, but the two trees side by side show the two revisions being compared. If a class exists in one revision but not the other this is highlighted.

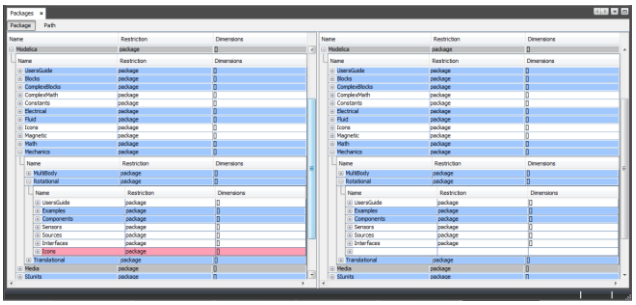

Figure 2: Packages View showing revisions of Modelica Standard Library

3.2 Class Views

An individual Modelica class can be viewed in a number of ways, each designed to visualize a different aspect of the model. Like the global views, these compare the revisions side by side.

3.2.1 Component Structure View

The Structure view shows the components declared in the class. Any components existing in one and not the other will be highlighted, as will differences between individual components. These differences could be the type, dimensions, value or other modifications on the component.

3.2.2 Package Structure View

The Package view shows a tree view of classes declared below the class. Any classes existing in one and not the other will be highlighted. Differences in attributes such as class restriction, dimensions or replaceability will also be highlighted.

3.2.3 Inheritance View

The Inheritance view shows a tree view of classes that the class extends from.

3.2.4 Dependencies View

The Dependencies view shows a tree view of all classes that the class depends on. This could be via inheritance or use as a component. By expanding the tree the user can quickly tell if any dependencies have changed, and changes to key attributes for each class are also highlighted.

3.2.5 Annotations View

In most cases the user will wish to ignore annotations, as these do not affect simulation results. However an Annotations view is included, which shows the user a tree view of all the annotations attached to the class and to components within it.

3.2.6 Instantiated Components View

The Instance view is a powerful way of comparing revisions of a class. The class is shown as a tree of components, both locally declared and inherited. Below each component in the tree is the set of components generated by the flattening of the component.

This allows the user to visualize changes in the resulting flattened model.

3.2.7 Equations View

The Equations view shows a tree with the equations for the class, this can be expanded to see equations from all components in the model, matching the Instance view for components. Equations are determined after modifications are applied but are not simplified and connection equations are not elaborated.

3.2.8 Code View

The Code view is a traditional code diff view, showing the code for each revision with changed lines highlighted.

3.3 Navigation

It is important to be able to quickly navigate through the packages in order to locate a source of a change. Within the global Packages view clicking on any class opens a view of that class. Clicking on a reference to another class within any view, such as Inheritance or Dependencies views, opens a view of it.

Navigating to a different view on the same class is as simple as selecting the tab for the required view.

4 Application Examples

The following examples are to illustrate simple cases where a change that appears significant when comparing code directly causes an insignificant change to the model results, or where a change that appears insignificant comparing code causes a significant change in the model results.

4.1 Editing Diagram

When building Modelica models in a Modelica environment some of the most common operations are in the diagram editor. Some of these have an effect on the results of the model, such as adding or removing components or connections, while some have no effect.

Moving a component in the diagram layer will result in the Placement annotation for that component changing, and the Line annotation for any rerouted connections changing, therefore creating differences on several lines within the model definition. This will be viewed within a VCS client or browser as a significant change, however within Converge it is only shown in the Annotations and Code views.

4.2 Refactoring Inheritance

A common change to models that a library developer may make is moving connectivity and common variables from a model to a partial class, and changing the model to extend from the partial class. This then allows other models to inherit the same connectivity.

```
package Springs
  model SimpleSpring
      Flange_a flange_a;
      Flange_a flange_b;
      Force f;
      Position s_rel;
      parameter Stiffness k=1000
"Spring rate";
      parameter Distance s_rel0=0 "Un-
stretched spring length";
  equation
      s_rel = flange_b.s - flange_a.s;
      flange_b.f = f;
      flange_a.f = -f;
      f = c*(s_rel - s_rel0);
  end SimpleSpring;
end Springs;
```

Code 1: Package before Inheritance Refactoring

```
package Springs
  partial model PartialSpring
       Flange_a flange_a;
       Flange_a flange_b;
       Force f;
       Position s_rel;
  equation
       s_rel = flange_b.s - flange_a.s;
       flange_b.f = f;
       flange_a.f = -f;
  end PartialSpring;

  model SimpleSpring
       extends PartialSpring;
       parameter Stiffness k=1000
"Spring rate";
       parameter Distance s_rel0=0 "Un-
stretched spring length";
  equation
       f = c*(s_rel - s_rel0);
  end SimpleSpring;

  model SimpleSpringDamper
       extends PartialSpring;
       Velocity v_rel;
       parameter Stiffness k=1000
"Spring rate";
       parameter Damping d=10 "Damping
rate";
       parameter Distance s_rel0=0 "Un-
stretched spring length";
  equation
       v_rel = der(s_rel);
       f = c*(s_rel - s_rel0) +
d*v_rel;
  end SimpleSpringDamper;
end Springs;
```

Code 2: Package after Inheritance Refactoring

A change such as this will be viewed within a VCS browser, and within the Packages, Class Structure and Inheritance views of Converge, as a significant change. A new class has been added, and the original class has had components removed and an extends-clause added.

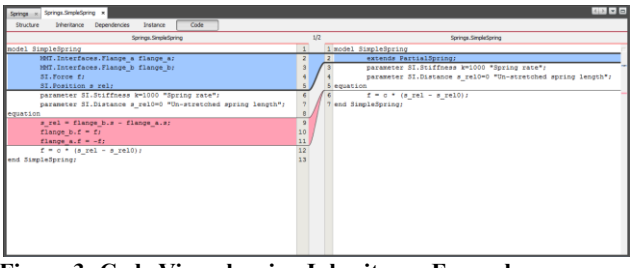

Figure 3: Code View showing Inheritance Example

However, by using the Instance view within Converge, it can be seen that the resulting set of variables has not actually changed. So the user can identify using Converge that this change should not have an impact on the simulation results.

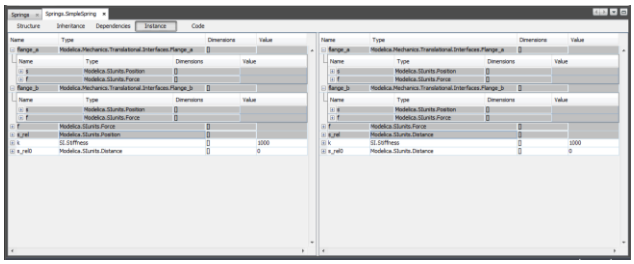

Figure 4: Instance View showing Inheritance Example

4.3 Changing Imports

Changing import statements in a class can change the types of components, cause an error or have no effect at all. Viewed in a traditional diff tool only the line containing the import statement would be highlighted as a change.

The following two models are the same except for a change of import statement. However the components and equations below the “spring” component are different.

```
  model MySpringAndMass
       import Spring =
Springs.SimpleSpring;
       Spring spring;
       Mass mass;
       Ground ground;
  equation
       connect(ground.flange,
spring.flange_a);
       connect(spring.flange_b,
mass.flange_a);
  end MySpringAndMass;
```

Code 3: Model before Import Change

```
  model MySpringAndMass
      import Spring =
Springs.SimpleSpringDamper;
      Spring spring;
      Mass mass;
      Ground ground;
  equation
      connect(ground.flange,
spring.flange_a);
      connect(spring.flange_b,
mass.flange_a);
end MySpringAndMass;
```

Code 4: Model after Import Change

Using the Instance view within Converge, it can quickly be seen what the resulting difference is between the models. Because Converge can resolve the types of components within the model it takes account of the import statements when determining the component tree.

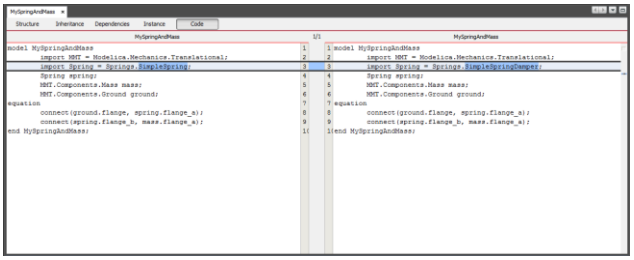

Figure 5: Code View showing Import Example

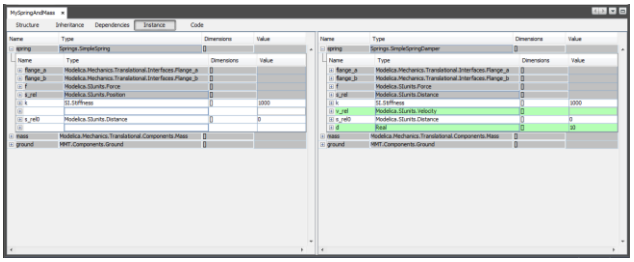

Figure 6: Instance View showing Import Example

4.4 Redeclarations

A redeclaration of a component is a small change to the Modelica code that can have a significant change to the flattened model. The spring and mass example from above can be restated as a redeclaration as follows.

```
  model MySpringAndMass
      replaceable Springs.SimpleSpring
spring constrainedby
Springs.PartialSpring;
      Mass mass;
      Ground ground;
  equation
      connect(ground.flange,
spring.flange_a);
      connect(spring.flange_b,
mass.flange_a);
  end MySpringAndMass;
```

Code 5: Spring mass model with replaceable spring

```
  model MySystem
    MySpringAndMass springMass;
  end MySystem;
```

Code 6: Model before redeclaration

```
  model MySystem
    MySpringAndMass spring-
Mass(redeclare
Springs.SimpleSpringDamper spring);
  end MySystem;
```

Code 7: Model after redeclaration

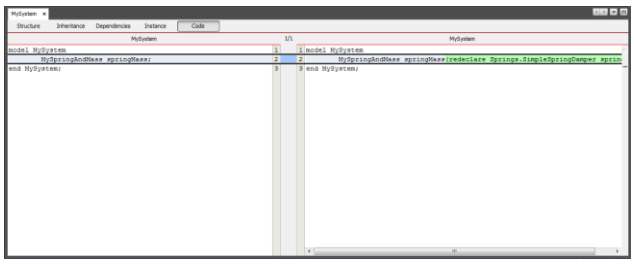

Figure 7: Code View showing Redeclaration Example

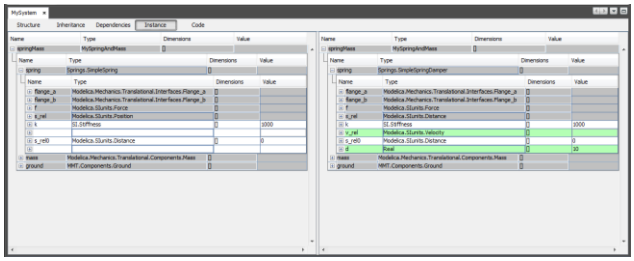

Figure 8: Instance View showing Redeclaration Example

5 Implementation

The ability to load, analyze and navigate the structure of Modelica models, including their instantiated form, is provided by the Modelica SDK [14].

The Modelica Specification [15] defines the mapping between a Modelica package hierarchy and a filesystem. For the general case this is only applicable to a directory structure or a Modelica archive file. The Modelica SDK has an interface that represents this mapping. This defines the methods required to access a hierarchy of Modelica files. Within Converge, implementations of this interface are provided that communicate directly with the VCS system, allowing Modelica models to be loaded for a specified revision without performing a "checkout" operation.

5.1 Status and Limitations

Currently Converge works with Subversion (SVN) repositories. This is being expanded to include a range of version control systems.

Converge is based on a Modelica implementation designed for Modelica 3.x. Libraries containing experimental language features, especially those that change the general syntax or class look-up process, may not give the expected results.

Differences between connections and overconstrained Connections.branch/.root statements are shown in the tool. Connection equations are not generated, including Stream equations and overconstrained branches.

5.2 Issues

It should be stressed here that the analysis of whether a change to a model potentially affects the results has to make the assumption that a change to an annotation has no such effect. Since the introduction of the Embedded Systems section to the specification this is no longer the case, as the mapping to the target system is defined as an annotation.

5.3 Future

Although Converge is not attempting to compile or simulate models, it can still detect sources of errors. Examples of these are changes of names or removal of classes or components that result in failure to find a class or component, or some cases of incompatible types or dimensions. If such an error occurs in the working version but not in the "head" revision within the VCS then a warning could be issued to the user.

6 Conclusions

In this paper, we have introduced a tool, Converge, for comparison of revisions of Modelica packages within a version control system. This will allow users to understand the development of models over time and to solve problems caused by changes in dependent Modelica libraries.

References

[1] Modelica, http://www.modelica.org

[2] Collins-Sussman, B., The Subversion Project: Building a Better CVS, Linux Journal, Volume 2002 Issue 94, February 2002

[3] Morse, T., CVS, Linux Journal, Volume 1996 Issue 21, Jan. 1996

[4] GIT - Fast Version Control System, http://git-scm.com

[5] O'Sullivan, B., Distributed revision control with Mercurial, Mercurial Project 2007

[6] TortoiseSVN, http://tortoisesvn.org

[7] Visual Studio, http://www.microsoft.com/visualstudio/

[8] Eclipse, http://eclipse.org

[9] Netbeans, http://netbeans.org

[10] Trac, http://trac.edgewall.org

[11] Modelica Association Trac Instance, http://trac.modelica.org

[12] Christen, E., Bakalar, K., VHDL-AMS, a hardware description language for analog and mixed-signal applications, Circuits and Systems II: Analog and Digital Signal Processing, Volume 26 Issue 10, 1999

[13] Converge, http://www.deltatheta.com/products/converge/

[14] Harman P., Tiller M. Building Modelica Tools using the Modelica SDK, Modelica 2009

[15] Modelica Language Specification, Version 3.2, Modelica Association 2010