Proceedings
of the 3rd International Modelica Conference,
Linköping, November 3-4, 2003,
Peter Fritzson (editor)

Michael Tiller
Ford Motor Company:
Parsing and Semantic Analysis of Modelica Code for Non-Simulation Applications
pp. 411-418

Paper presented at the 3rd International Modelica Conference, November 3-4, 2003, Linköpings Universitet, Linköping, Sweden, organized by The Modelica Association and Institutionen för datavetenskap, Linköpings universitet

All papers of this conference can be downloaded from
http://www.Modelica.org/Conference2003/papers.shtml

Parsing and Semantic Analysis of Modelica Code for Non-Simulation Applications

Michael Tiller
Powertrain Research Department, Ford Motor Company

ABSTRACT

While most discussions involving Modelica focus on its technical capabilities (*i.e.* object-oriented modeling, handling of DAEs, standard libraries, *etc.*), the benefits of having a formal specification of the language syntax and semantics for non-simulation applications are often overlooked. Unlike many proprietary modeling technologies, where the syntax and semantics of the models change according to the whims of the tool vendor, the syntax and semantics of Modelica models are clearly spelled out in the Modelica specification and considerable effort is made to maintain backward compatibility while adding new capabilities to the language. Not only does this allow vendors to develop simulation environments that independently support a common language, it also allows for the development of ancillary tools to support the model development process. Recognizing some of the best practices in software development, this paper discusses a set of utilities used to analyze existing Modelica models and provide feedback on the structure of the models. These analyses can highlight problematic or unused code, check that code is compliant with specific style guidelines or generate "intelligent" reports on differences between different versions of a model.

1 Motivation

For years, Ford Motor Company has been developing several proprietary Modelica libraries. While we have a talented team of developers and we meet on a regular basis to discuss the evolving structure of our model libraries, it is still difficult to contain the "entropy" that develops due to code fragments that are no longer actively maintained.

After many years focusing on development, it was necessary to take a step back and consider how to manage the growing complexity of our model libraries. Recognizing the common challenges between software development and model development, we have always tried to leverage the best practices from software engineering and incorporate them into our model development. For example, we use a version control system internally to manage releases of our model libraries and we have a web-based issue tracking system that we use to log bugs and feature enhancements. However, these capabilities were easy to leverage because of the availability of general-purpose, out-of-the-box tools (*e.g.* CVS).

Unfortunately, there are many code analyses that we would like to perform that are not supported by general-purpose software engineering tools because they require language specific information. Furthermore, existing Modelica tools focus mainly on simulation-oriented capabilities. As a result, we decided to implement our own utilities to assist us in maintaining our code base.

2 Syntax and Semantics

2.1 Introduction

This section will discuss the steps, tools and ideas involved in taking Modelica code as it appears in a file and creating a representation that captures the underlying "meaning" (*e.g.* type, baseclasses, scope) of the various structural entities.

It should be noted that the analysis capabilities described in this paper do not implement and/or check all the semantics defined in the Modelica specification. Instead, they assume that the code is legal Modelica code generated by a tool (*e.g.* Dymola) that conforms to the Modelica specification. Ideally, we hope that our semantic processing may eventually encompass all the semantics discussed in the Modelica specification but fortunately the analyses described in the paper do not require a complete implementation, only the capability to definitively resolve the types of entities during instantiation.

2.2 Tools

Before presenting additional details about the individual steps involved in processing Modelica code, it is useful to include some discussion of ANTLR [1], the tool used to automate the process of parsing Modelica code. The ANTLR toolset can generate software objects for performing

lexical analysis, grammar parsing and tree parsing (these tasks will be discussed in detail in the remainder of this section). ANTLR includes several useful features including:

- Java, C++ and C# as target languages
- Portable and readable generated code
- Automatic syntax tree construction.
- Active community
- Ongoing development

A surprisingly common question people ask is "Why was Modelica developed with its own unique grammar? Why not simply use XML to describe the format of Modelica files?" Indeed, the wealth of available eXtensible Markup Language [2] (XML) parsers and tools [3] would make the parsing of Modelica files almost trivial. Terrence Parr, author of ANTLR, has provides an excellent discussion of this question in his essay "Humans should not have to grok XML" [5]. The short answer is that XML only addresses the issue of syntax, not the meaning of the constructs themselves. Furthermore, XML is best applied to file formats that are automatically read and written by computers not humans. It is for these reasons that the vast majority of programming languages (*e.g.* Java, C++, Haskell, C#, Python, Perl and Tcl) choose to define their own unique syntax (that is intuitive to human readers and writers) while only a handful of languages like XSLT [4] employ XML syntax. Viewed in this way, the approach taken when developing Modelica is completely consistent with how programming languages, in general, are developed.

That being said, a very compelling argument can be made for using XML to represent data structures needed by or resulting from semantic processing [6]. For example, one tool could be responsible for reading the Modelica code and generating an XML representation of the abstract syntax tree. Such a file could then be read by other tools and transformed into representations of instantiated models, hybrid differential-algebraic equations and pseudo-simulation code, *etc*. Such an approach would allow a clean partitioning of tasks and formal description of the various intermediate representations (*i.e.* using Document Type Definitions (DTDs) or XML Schemas).

2.3 Lexical Analysis

The first step in our process to uncover the meaning in Modelica code is to break the code into "tokens". Conceptually, tokens are the words that exist in Modelica (*i.e.* strings of characters delimited by whitespace). It is very easy to identify the tokens in a given file, but it is also necessary during this step to classify these tokens. Some tokens are easily recognized as keywords (*e.g.* `replaceable`, `parameter`, `final`). Other categories of tokens include literals (*i.e.* integers, reals, strings and Boolean values), punctuation (*i.e.* semicolons, periods, parentheses, *etc.*) and so on. Section 2.1 of the Modelica specification discusses the categories of tokens involved and the patterns used to recognize them. Using ANTLR, our lexical specification for Modelica required 12 non-trivial rules to identify tokens.

2.4 Grammar Definition

Previously, lexical analysis was described as the process by which "words" are extracted from Modelica code. Extending this analogy, grammatical analysis is the process of constructing meaningful "sentences". These sentences can describe definitions of new Modelica types, declarations of components or variables in a class, equations, modifications and so on.

Just as with lexical analysis, the patterns used to describe the grammar of the Modelica language can be found in Section 2.2 of the Modelica specification. An important aspect of creating or processing a grammar definition is avoiding any potential ambiguity. When described using an LL(k) grammar (as required by ANTLR), it is necessary for the parser to look two tokens ahead in order to resolve any ambiguities.

Using ANTLR, our description of the Modelica language involved 35 tokens (and their associated regular expressions), 70 rules and 32 fundamental node types.

2.5 Syntax Trees

2.5.1 Tree Construction

While processing lexical tokens and matching them to grammatical rules, ANTLR includes features to automatically generate a syntax tree to represent the underlying structure of the file being parsed. During tree construction, the goal is to filter out tokens that are only of syntactic significance (*e.g.* semicolons, which only exist to explicitly terminate certain structures) and preserve information that is necessary to fully understand the intent of the code. ANTLR provides a shorthand notation for tree construction that is very convenient, but there are still a few common operations that lack a shorthand representation.

2.5.2 Data Structures

ANTLR builds trees out of nodes and then associating these nodes through child and sibling relationships. By default, ANTLR assumes these nodes are homogenous (*i.e.* they are all of the same type in the target language). This approach works well for "text-to-text transformation" applications (where specific patterns of nodes are simply transformed into other patterns of nodes without a lot of semantic information). However, if the nodes in the resulting tree are likely to have a wide range of different types of information and/or methods associated with them, it is possible to instruct ANTLR to use specific node types (in the target language) for specific structural entities in the tree. The result is a heterogeneous tree structure. As mentioned in Section 2.4, the resulting trees are composed of 32 fundamental node types.

One of the advantages of using heterogeneous node types is the ability to "promote" entities that would normally be tokens into member data associated with that node. For example, Modelica definitions **must** include the name of the class being defined. One approach would be to store this name token as a child node of the definition node in the constructed tree. However, since this is an element that is always present, you can save some complexity in the tree structure (and some lookup time during processing) by storing this information directly as just a string in the definition node itself (as opposed to a child node). We use heterogeneous trees and reserved the use of child and sibling nodes for those structures that are variable (*i.e.* elements whose presence is not known *a priori*).

2.5.3 Tree Walking

ANTLR includes support for creating tree walker objects. Such "tree grammars" are typically much simpler than the formal grammar because they do not include strictly syntactic elements like punctuation and keywords. While tree parsers can be quite useful, we have chosen to use a more programmatic approach for most of the analysis. Rather than walking the tree, most of our analyses involve searching the tree structure for specific elements and then performing operations on those elements. The one case where we currently employ a tree parser is as a validator for our generated tree. By constructing the tree grammar we expect as a result of tree construction, we can apply that tree parser to any tree available (either from directly parsing Modelica code or resulting from programmatic manipulation of an existing tree structure) and identify any structures not described in the tree grammar. This is analogous to using a DTD or XML schema to validate an XML file.

2.6 Semantic Analysis

As mentioned previously, we assume that all code being parsed is syntactically and semantically legal. In this way, we can avoid implementing the complete semantics of the Modelica specification. Nevertheless, it is still necessary to implement many of the semantics in order to understand what is implied by the code. Without this knowledge, it would be impossible to perform the analyses described in Section 3.

The semantics in the Modelica specification [7] cover all aspects of the language necessary to translate a Modelica model into a system of hybrid, differential-algeabraic equations (DAEs). Fortunately, for non-simulation applications only a handful of these semantics are required. Specifically, we have implemented a set of semantics that allows us to instantiate all the components in a model (even those affected by redeclarations). We have neglected all semantics associated with equations and algorithms. As a result, the main task required as part of this instantiation is name lookup as described in Section 3.1 of the Modelica specification.

2.7 Issues

While creating these tools, there were several issues that we uncovered both in both the Modelica specification and ANTLR that are worth mentioning.

2.7.1 Modelica

In Modelica, comments are lexically significant but not grammatically significant and this can make the preservation of comments while rewriting Modelica code a challenge. One way to address this situation would be to make comments grammatically significant. Given the availability of descriptive strings for documentation purposes in Modelica, comments are really only necessary for "commenting out" definitions, declarations, equations or algorithmic statements. As such, they could be inserted as elements in the grammatical rules for those entities. While this would constrain the situations where comments could be used, it would make their preservation much simpler.

In addition, there are some features described in the Modelica specification that have never been implemented. Examples of such features include

the `within` statement and the `analysisType()` function. If a feature goes unimplemented for several years, it is probably worth revisiting that feature to see whether it is truly necessary or desirable. Weeding rarely used or unnecessary features out of the language helps minimize the work associated with developing parsing and semantic analysis tools which, in turn, makes Modelica easier to adopt.

Finally, there are a handful of rules in the Modelica grammar that make the task of resolving ambiguities difficult. Specifically, the use of "initial" as both a keyword and a function name is problematic since the same string, '`initial`', can fall into two different token categories (and this depends on where it appears grammatically). Another example of this kind of problematic "reuse" is the '`end`' string which can be used to close a long definition or appear as an element in an expression. Once again, this ambiguity presents a burden for the parser developer.

2.7.2 ANTLR

We chose to generate heterogeneous trees while processing the Modelica grammar. While ANTLR supports heterogeneous trees, using them with C++ as the target language presented many problems. For example, a bug in the garbage collecting mechanism of the AST base classes appears when using heterogeneous trees. In addition, even though ANTLR allows node types to be associated with specific tokens, this applies only during creation of the nodes. When they are referenced from within a rule, a cast is necessary. It is worth mentioning that C++ language support for heterogeneous node types in ANTLR are relatively new. All things considered, these are only minor annoyances and hopefully future versions of ANTLR will include improved support for heterogeneous AST construction.

3 Analyses

Most of the analyses described in this section require that models can be instantiated according to the instantiation process described in the Modelica Language Specification. As a result of this process, a syntax tree is generated to represent the structural elements of the instantiated model. It is then possible to conduct an analysis of the model by "walking the tree" looking for certain patterns and/or performing specialized calculations. This section discusses several specific types of analyses that are applicable to Modelica code.

3.1 Simple Metrics

The idea of "software metrics" has been around for many years [8]. We will begin our discussion with a few simple code metrics that can also be found in non-modeling contexts.

3.1.1 LOC

A common metric in software engineering is "lines of code" (LOC). While easy to measure, the metric itself is normally not that meaningful. For our purposes, we will count lines in each non-package definition and tally these lines for each package. Furthermore, we will define a "line" as any statement that ends in a semicolon. In other words, since line feeds and carriage returns are not grammatically significant, we will focus on the number of statements which is roughly equivalent to the number of lines.

3.1.2 Restricted Class Breakdown

Another statistic that is easy to collect but not very meaningful, is the breakdown of definitions by restricted class (RCB). This metric mainly serves how heavily utilized each restricted class type is within a given package hierarchy. This metric is similar to lines of code because it measures the "volume" of the code but does not accurately assess its complexity.

3.1.3 Inheritance Complexity

A more useful metric (and one that requires implementing instantiation semantics) is quantifying inheritance complexity. Inheritance complexity is a reflection of how confusing the use of inheritance would be to a user. While inheritance is useful for promoting reuse and avoiding the maintenance issues associated with redundant code, it can also make it difficult for users to understand the complete details of a model. Ideally, inheritance should be restricted to definitions that are:

- Used often – Definitions that developers are likely to be familiar with them.
- Necessary – To avoid base classes that introduce unnecessarily fine distinctions.
- Minimal – To keep the number of classes that developers must be familiar with to a minimum.
- Easily resolved – Modelica features such as replaceable types, dynamic scoping and lookup in enclosing scopes can make it hard for developers to easily figure out or remember what the base classes really are.

The inheritance complexity (IC) is computed as follows[1]. First, it is assumed that a definition that does not extend from another definition has an IC value of 1. For each extends clause, various adjustments are made to this score. If the definition being extended is used by fewer than 10 definitions, the IC is incremented by 1. If the definition being extended is used by fewer than 5 definitions, the IC is incremented by an additional 1. If the definition being extended contained less than 3 declarations and less than 3 equations then the IC is again incremented by 1. The IC value for the definition being extended is then multiplied by a scaling factor and added to the IC for the current definition. If the type being extended is replaceable and locally defined, the scale factor is 2. If the type is replaceable but defined outside the scope of the current definition, the scale factor is 3. Finally, if the definition being extended is declared outer, the scale factor is 2.

3.2 Style Guidelines

Looking beyond simple metrics, another type of analysis is to check for conformance to style guidelines. Style guidelines are formulated to promote reusability and consistency of code and many of these style guidelines can be formulated in such a way that they can be automatically verified. Any definitions that contain non-conforming code can be identified in automatically generated reports.

At Ford, we have an extensive set of style guidelines. In this section, we will preset a few of these guidelines, discuss why these guidelines were adopted and explain how we automatically check for conformance.

3.2.1 Naming Conventions

According to our style guidelines, all Modelica definitions must begin with a capital letter while declarations must begin with a lower case letter unless they contain only a single letter in which case they should be capitalized. This rule was adopted because it makes it easy to recognize whether a fully qualified name corresponds to a type or an instance.

To check naming conventions, we visit each definition in memory and process the list of enclosed definitions and declarations looking for non-conforming names.

[1] This is just an initial algorithm to demonstrate how such a metric could be calculated. With time, a better algorithm could probably be developed.

3.2.2 Documentation

For a model library to be generally useful, it is important for model libraries to be well documented. Using the tools described in this paper, we are able to automatically review all definitions and declarations and check for the existence of documentation annotations. Futhermore, this analysis can check to see if descriptive strings have been associated with each definition and declaration so that generated GUI dialogs include additional useful information.

3.2.3 Mixing Equations and Components

The last guideline we will discuss is a restriction against representing behaviour both textually and graphically in the same model. To accomplish this, we must classify each declaration as either textual or graphical. For the purposes of this analysis, connector definitions that appear graphically are ignored. The point of this guideline is to avoid confusion that can develop when trying to grasp the behaviour of a model when aspects of that behaviour span both the text layer and the diagram layer.

As of Dymola 5.x [9], it has been possible to quickly assess this restriction visually by inspecting the Modelica source layer. By default, everything that appears in the diagram layer is filtered out. As such, if you see equations and graphical icons in the Modelica source, the definition you are viewing violates this rule. Nevertheless, visual inspection for entire model libraries is not practical and that is the motivation behind having a tool capable of automatically and exhaustively checking an entire library.

3.3 Coverage Analysis

3.3.1 Background

The most elaborate analysis possible with our tools is what we call "coverage analysis". For each of our model libraries (*i.e.* libraries composed of component, subsystem or system model), we try to maintain a companion test suite library. The goal of the test suite library is to include tests of every model in the model library.

These test suites are useful for several reasons. First, they provide us with a way to assess whether recent bug fixes and/or enhancements to our model library have not corrupted any of the models. In addition, we perform similar checks across tools or tool versions. Finally, we can analyze the test suite library identify any coverage gaps (*i.e.* any components that are not tested).

3.3.2 Analysis Algorithm

The first issue that must be addressed is which models to apply the analysis algorithm to. Stated another way, which models are the test cases? Some rather obvious criteria are:

- Any model in a test suite library.
- Any model that extends from certain base classes (e.g. `extends TestCase;`).
- Any model that does not contain connectors.

Of these, the last criteria is the most general and requires the least discipline on the part of the test suite developer. However, because of the time required to conduct the analysis and the large amount of potential data generated as a result, it may be desirable to use one of the more restrictive criteria. Regardless of the criteria chosen, the algorithm is the same.

The first step in the process is to instantiate each test case. Although the complete instantiation process is described in detail in the Modelica specification, the basic principle is to construct the component tree for each model (factoring in redeclarations, base classes, *etc.*). As a result of it, it should be possible to identify the type of every instantiated component. The set of instantiated types is recorded as each test case is instantiated.

When every test case has been instantiated, you are left with the set of all types that were instantiated by at least one test case. You can then iterate over the set of all type definitions in your model library and check to see if they are in the set of instantiated types. Any definition that was not instantiated represents either a gap in coverage by the test suite or a definition that should be deprecated.

Coverage analysis is a good way to make sure that your model library doesn't contain any unused or unnecessary definitions. It also provides feedback on whether a given test suite provides accurate coverage.

4 Results

4.1 Running the Analysis

Normally, the use of our models is scattered over a number of different packages. Obviously, we would like to have a complete test suite that exercises every single model we have. A more reasonable near-term goal would be that every model is used in one of the many packages (most of them application specific) that we have developed.

To support this possibility, the command line syntax of our tool requires the first argument to be the package being analyzed and all other arguments are assumed to be packages that may potentially use components in the first package. A typical command line invocation might look something like:

```
% Metrics Ford FordTestSuite AppLib1 … AppLibN
```

4.2 Sample Library Results

To demonstrate the results that are generated from our tool, consider the sample package shown in Figure 1. The details of the models are not particularly meaningful for the purposes of evaluating the metrics for the code. Running our `Metrics` program tells us that the library includes 3 models, 1 type definition and 1 package. For a simple package like the one shown in Figure 1, this is obvious. These kinds of statistics are interesting for larger packages where counting definitions becomes impractical. While we will get to additional metrics in subsequent sections, for now let us focus on coverage analysis. Assume we use the package in Figure 2 as our set of regression tests for package in Figure 1. The results of the analysis are shown in Table 1.

```
package CompLib "Component Library"
  model A "Simple model"
    Real x;
    annotation(
      Documentation(info="Simple model"));
  equation
    der(x) = 2.3*time;
  end A;
  model B "Typical model"
    type GrowthRate = Real(min=0);
    Real x;
    parameter GrowthRate c=2.3;
  equation
    if time<1.0 then
      der(x) = c*time/2;
    else
      der(x) = c*time;
    end if;
  end B;
  model C "Detailed model"
    Real x, y;
    parameter Real Alpha=0.1, Beta=2;
    parameter Real Gamma=4, Delta=0.4;
  equation
    der(x) = Alpha*x*y-Beta*x;
    der(y) = Gamma*y-Delta*x*y;
  end C;
end CompLib;
```

Figure 1: Sample Component Library

```
package CompTestSuite
  import CompLib.*;
  model System1
    A a1, a2;
  end System1;
  model System2
    A a;
    C c;
  end System2;
end CompTestSuite;
```

Figure 2: Sample Test Suite

Definition Name	Times Used	Is Documented
CompLib.A	3	Yes
CompLib.B	0	No
CompLib.B.GrowthRate	0	Not Applicable
CompLib.C	1	No

Table 1: Sample Coverage Analysis

4.3 Ford and Modelica Libraries

We thought it would be interesting to compare the metrics of our proprietary Ford powertrain library with the Modelica standard library. For the purposes of this analysis, only the examples in the Modelica standard library were used. The results from this analysis are shown in Figure 3. The X-axis in each plot lists a series of categories and the Y-axis indicates the percentage of definitions in each library that fall into that category.

The documentation and naming convention metrics cannot be applied to `type` definitions. That is why, for each of these metrics, two sets of results shown. One set includes the all possibilities while the other set only considers the cases where the metric can be applied meaningfully. This highlights the number of type definitions in the Modelica library (e.g. `Modelica.SIunits`).

Some interesting results found in Figure 3 are:

- Nearly all the models in both libraries are represented by either strictly textual or strictly graphical information.
- Over 70% of the Ford library isn't covered by a test case.
- The biggest difference between the libraries in the documentation. About 90% of the definitions that can be documented in the Ford library do not include documentation while this is true for less than 40% of the definitions that can be documented in the Modelica standard library.
- Naming convention compliance is surprisingly similar for the libraries.

5 Future Applications

The analyses described in this paper are just a few of the many non-simulation related tasks that can be automated with an appropriate library for parsing and processing Modelica code. Other potential applications could include command-line compilers, "lint" like analysis for undesirable construct, pretty-printing tools, ETAGS generators for Emacs, intelligent differencing tools and so on. Although unimplemented, these tasks further justify the utility of such capabilities. Rather than discuss each of these detail, we will present one example in some detail.

5.1 Obfuscation and Filtering

So far, none of the analyses that have been discussed involved rewriting Modelica code. However, for reasons related to protecting intellectual property, it is quite likely that developers of Modelica code may wish to somehow obfuscate or remove certain sensitive models. Note that even with tools capable of encrypting Modelica models, there may still be a need for obfuscation (*e.g.* exporting models to a Modelica tool or environment that doesn't support encryption).

The most extreme course of action would be to filter models out. Another more moderate approach would be to obfuscate models so that they functioned properly but were hard to understand. To filter models, it would only be necessary to remove their definitions from an existing tree structure before writing that tree structure back out as Modelica code.

Obfuscation is a bit more difficult to implement. The first step would be to identify which definitions needed to be obfuscated (*e.g.* using a special annotation) and then which elements of that definition were impacted (*e.g.* only protected elements). For the elements to be obfuscated, several actions are possible programmatically. First, you would almost certainly want to strip off any descriptive strings. Second, for real variables you would probably change their type to `Real` rather than something that hinted at their units. Finally, you could change the names of these elements so that their names did not hint at their meaning. This last requirement is very tricky because it would require changing any references to the previous name.

6 Conclusions

While the emphasis in most Modelica applications is on modeling, as Modelica becomes

used for "enterprise scale" activities it will be increasingly necessary to have tools capable of analyzing the quality of the underlying code. This paper highlights several practical analyses that are currently in use and several other potential analyses that could be facilitated by such tools.

7 Acknowledgments

Peter Aronsson and Peter Fritzson from PELAB at Linköping University provided me with the source code for their Open Source Modelica project. Although I did not use the code directly, I it was useful as a reference in developing the tools discussed in this paper.

Adrian Pop, also from PELAB, has done considerable work in understanding the role of XML in processing Modelica code. His work discusses the ideas about XML presented in Section 2.2 in greater detail.

Finally, I would also like to thank Hans Olsson at Dynasim AB for helping to explain, in implementation terms, the details described in the Modelica specification.

8 References

1. T. Parr, "ANTLR 2.7.2 Reference Manual", http://www.antlr.org/doc/index.html
2. "Extensible Markup Language (XML) 1.0, Second Edition", *World Wide Web Consortium*, http://www.w3.org/TR/REC-xml
3. L. M. Garshol, "XML tools by name", http://www.garshol.priv.no/download/xmltools/name_ix.html
4. "XSL Transformations Version 1.0", *World Wide Web Consortium*, http://www.w3.org/TR/xslt
5. T. Parr, "Humans should not have to grok XML", http://www-106.ibm.com/developerworks/library/x-sbxml.html
6. A. Pop, P. Fritzson, "ModelicaXML: A Modelica XML Representation with Applications", *Modelica'2003 Conference Proceedings*.
7. "Modelica Language Specification, Version 2.0", *Modelica Association*, 2002, http://www.modelica.org/documents/ModelicaSpec20.pdf
8. C. Jones, "Applied Software Measurement : Assuring Productivity and Quality," *McGraw Hill*, 1991.
9. "Dymola User's Manual, Version 5.0a", *Dynasim AB*, Sweden, 2002

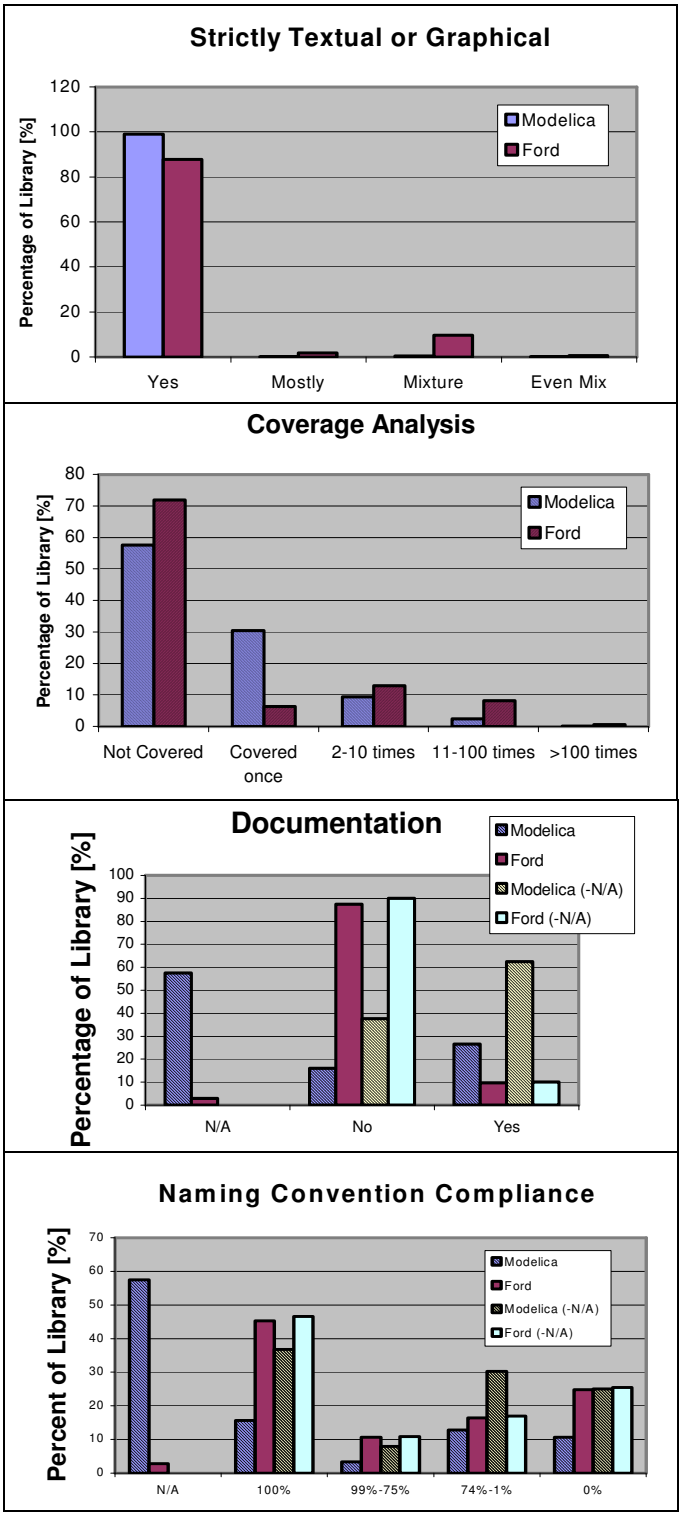

Figure 3: Comparing Ford and Modelica Libraries