A plan for building renaming support for Modelica

Hedin, Görel; Åkesson, Johan; Söderberg, Emma

2009

LUND UNIVERSITY

PO Box 117
221 00 Lund
+46 46-222 00 00

A Plan for Building Renaming Support for Modelica

Görel Hedin Emma Nilsson-Nyman
Dept. of Computer Science, Lund University
(gorel/emma)@cs.lth.se

Johan Åkesson
Dept. of Automatic Control, Lund University
jakesson@control.lth.se

Abstract

We discuss our current work on building an IDE for Modelica, and how we intend to support renaming. Our current implementation of the compiler and the name completion support is done using reference attribute grammars, implemented in the JastAdd metacompilation system. For renaming we plan to follow the approach of inverse lookups, developed by Schäfer, Ekman, and de Moor. Modelica has challenging naming semantics, providing a good test for this approach.

1. Introduction

The successful development of extended and new programming languages is currently hampered by the high cost of tool building. Developers are accustomed to the integrated development environments (IDEs) that exist for general-purpose languages, and demand the same services for new and experimental languages. We are working on lowering the cost for building IDEs by making use of declarative compiler technology. More specifically, we are using reference attribute grammars [7] as implemented in the JastAdd system [8, 4], and shown to work for full languages like Java [5]. This way of building compilers allows tools like IDEs to make use of computations defined by the compiler, and to extend those computations to fit the needs of the tools. In particular, the semantic services of an IDE, i.e., services that make use of various static program analyses, can benefit from this approach. Examples include cross-referencing [10], name completion [6], and renaming [12].

We are in the process of building several IDEs using this approach, and one of our case studies is an IDE for the language Modelica [1]. Modelica is a language for modelling physical systems by means of differential equations. It makes use of classes and inheritance for modularizing knowledge and has substantial standard libraries for different engineering fields, for example, electrical engineering, chemical engineering, mechanical engineering, etc. Also, embedded control systems modeling is supported. While having many similarities to general-purpose object-oriented languages like Java and C++, it also differs in many ways. For example, it has no runtime semantics. Instead, Modelica programs are used for analysis of physical models, using time simulation of the differential equations.

Modelica is primarily used to construct detailed mathematical models of the physical behavior of products or processes. Typically, such models are large, commonly in the range of 10,000 to 100,000 equations and variables. The Modelica language emerged as a result of the need to efficiently manage such models while at the same time promote model reuse and standardized model exchange. Once a model has been constructed, *virtual experiments* can be performed where the properties of the modeled system are assessed. This approach is referred to as *model-based engineering*. One of the main benefits of model-based engineering is that it enables engineers to explore system behavior early in the design process, even prior to construction of the product. While virtual experiments do not replace field tests, they can help reducing the need for expensive real world testing. Also, models enable design activities to be performed in parallel; a model of a physical plant can be used as a basis for developing a control system, while at the same time the real product is constructed. This methodology is used extensively, e.g., in the car industry.

Modelica has some specific language constructs that are challenging for name analysis, and thereby also for building semantic services like refactoring support. In addition to structural subtyping, multiple inheritance, and unlimited nesting of classes, it has a feature for *redeclaring* types, a construct somewhat similar to generic types.

Our work is done in the JModelica.org project on creating open-source extensible tools for Modelica [11]. Concerning semantic services, the JModelica.org IDE so far supports basic name completion, and the next step will be to build basic refactoring support, focusing on renaming. We plan to use the renaming approach based on *inverse lookups*, developed by Schäfer, Ekman, and de Moor [12], but adapted to the specific constructs of Modelica.

3rd Workshop on Refactoring Tools '09 Oct. 26, 2009, Orlando, FL

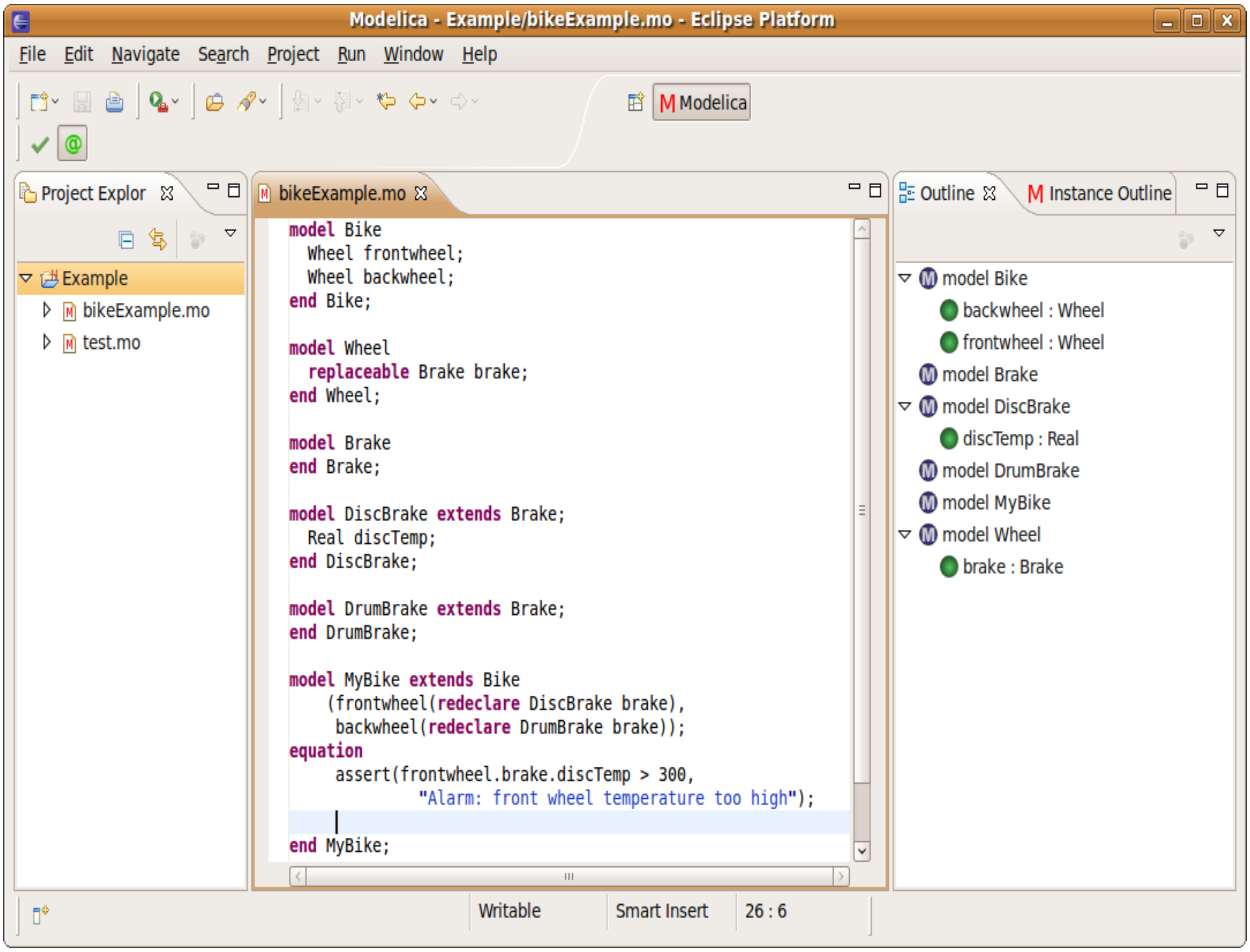

Figure 1. A Modelica example in the JModelica IDE

The rest of this position paper is structured as follows: In Section 2 we discuss name analysis for Modelica, and how the redeclare construct is handled, showing examples from our current Modelica IDE. In Section 3 we outline how we intend to adapt the inverse lookup approach to support renaming for Modelica. Section 4, finally, gives a few concluding remarks.

2. Name analysis in the JModelica IDE

Fig. 1 shows a screenshot from our IDE with some example Modelica classes, called *models* in Modelica.

In Modelica, a class can have *components*, which are compile-time instances of other Modelica classes. For example, the class `Bike` has two components of class `Wheel`. When declaring a component, its class may be modified by redeclaring certain internal components by more specific types. For example, the `Wheel` has a component `brake` of a replaceable type `Brake`. When declaring a `Bike` component, or a subtype like `MyBike` in the example, we can redeclare the `brake` component of its wheels to have a more specific type. In the example, we define the front wheel to have a `DiscBrake` and the back wheel to have a `DrumBrake`. This is similar to covariant generics in general-purpose languages, but does not bring about the same kinds of assignment compatibility issues, see e.g., [9], since components in Modelica are compile-time instances.

In class `MyBike`, an equation has been added which accesses the temperature of the front wheel's brake disc, in order to set an alarm when the temperature is too high. Note that while the `brake` of a `Wheel` is declared as a `Brake`, `MyBike` can treat it as a `DiscBrake` because of the redeclaration, and access the `discTemp` component.

To perform name analysis, consider finding the declaration of the `discTemp` in the expression

```
frontwheel.brake.discTemp
```

To do this, the redeclaration of the frontwheel brake needs to be taken into account. In the JModelica compiler, a com-

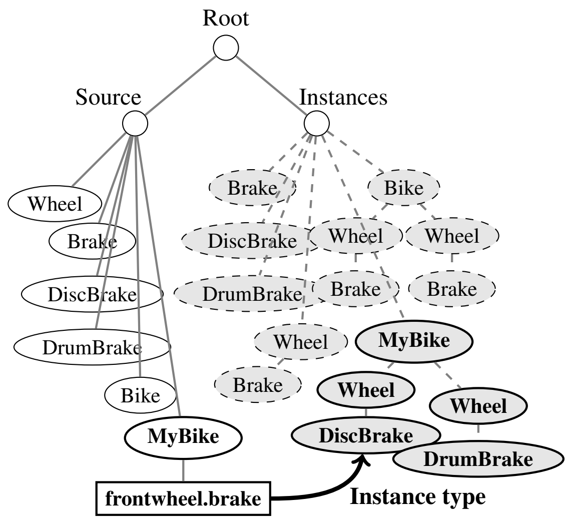

Figure 2. Structure of expanded AST, the source subtree to the left (white nodes) and the instance subtree to the right (grey nodes). Dashed lines indicate on-demand expansion of the AST, dashed nodes are unexpanded nodes. The instance node for **MyBike** has been expanded. The thick arrow points out the instance type of **frontwheel.brake** in **MyBike**.

ponent *instance tree* is built during compilation to simplify this problem. The abstract syntax tree (AST) is extended at compilation time with a component instance tree whose upper part is isomorphic to the class structure of the source code, and with lower subtrees corresponding to the unfolded component structure of those classes. See Fig. 2.

The instance AST is introduced mainly to handle *modification environments*, which consist of an ordered set of modification constructs, including redeclares as seen in the example code in Fig. 1. When a class is instantiated, a corresponding modification environment must be considered in order to take potential redeclarations into account. Notice also that the same class may be instantiated in several locations in different environments. Within the instance tree, modification environments are represented explicitly, which gives a transparent and efficient implementation. An instance node does not contain a complete copy of the corresponding source class, but only sufficient information for allowing name analysis. A link back to the source class is used for accessing information that is the same for all instances. See [2] for details on the construction of instance trees.

As an example, to perform name analysis inside `MyBike`, the AST will be expanded with an instance of `MyBike`, which in turn will contain instances of `frontwheel` and `backwheel` that have `brake` instances of the correct types (`DiscBrake` and `DrumBrake`). This allows the declaration of `discTemp` to be found from within `MyBike`. The instantiation tree is also used to compute name completion menus in the IDE.

3. Renaming in JModelica

Name analysis is implemented in the JModelica compiler using reference attribute grammars, applying the *lookup* technique developed for the JastAddJ Java compiler [3]. In this technique, each identifier access has an attribute that refers to the appropriate declaration. To define that attribute, parameterized *lookup attributes* are used that delegate the name analysis computation, for example, from local method, to class, to superclass, etc. In the JModelica compiler, the same general technique is applied to fit the particular constructs in Modelica and it is adapted to perform the lookup in the instance AST rather than in the source AST in order to handle redeclares.

To support renaming for Modelica, we plan to apply the *inverse lookup* technique, introduced by Schäfer, Ekman and de Moor [12]. This is a general technique for renaming, that extends the above mentioned lookup technique for name analysis. The key idea is to define *access* attributes that invert the lookup attribute. More precisely, for a program position p in the AST, the lookup attribute can be seen as a partial function from access to declaration:

$$lookup_p : access \rightarrow decl$$

The inverse of this relation, or the access computation, is:

$$access_p : decl \rightarrow access$$

where $access_p$ should be defined in such a way that the following correctness condition holds, for any position p and declaration d, and provided that $access_p(d)$ is defined:

$$lookup_p(access_p(d)) = d$$

This technique supports more general renaming than most other refactoring tools, in that accesses can be replaced by accesses with another syntactic structure. For example, renaming a field from `x` to `y` could cause a simple access `x` to be replaced by a qualified access `this.y`, to avoid a name conflict with a local argument named `y`.

To find endangered accesses when a declaration has been renamed from `x` to `y`, the technique used in [12] is to traverse the entire AST, and treat all simple accesses of `x` and `y` as potential candidates. This simple technique is reported to perform well in practice.

In applying this technique to our JModelica IDE, we will need to define inverse lookup functions that take the Modelica language constructs into account. Examples of relevant differences from Java include Modelica's use of structural subtyping, multiple inheritance, and the redeclaration feature. The JModelica compiler already includes name analysis support, i.e., lookup attributes, that take these constructs into account. We expect that the definition of the inverse lookup functions will be possible to develop in a similar way. As a first step, we will only support simple name changes of accesses, like in standard refactoring tools. We will then investigate how to handle replacing simple accesses by qualified ones, to make full use of the approach.

Even if the name analysis is performed in the instance AST, it will be important to do the computation of endangered accesses in the source AST. This is because the instance tree can potentially be extremely large, containing unfolded components of all classes, including those in libraries. For compilation, the size of the instance tree is not a problem because it is built on demand, and only the parts actually needed for name analysis will be expanded. To limit the traversal to the source AST, we will need to add reference attributes that link each component or class in the source AST to the corresponding instance in the instance AST. Because of the compile-time instantiation, these points are well-defined. The compiler already contains reverse links, linking each instance back to its corresponding position in the source AST.

4. Concluding remarks

We are building JModelica, an open-source compiler and IDE for the language Modelica. In this position paper we have sketched how we intend to extend the IDE with renaming support. A key challenge when implementing semantic services for Modelica is to handle the complex name analysis rules including structural subtyping, multiple inheritance, and type redeclaration. In the JModelica compiler we have solved this by performing name analysis in an instance tree, containing a compile-time unfolding of the program. The name analysis is implemented using reference attribute grammars, adapting a lookup attribute technique previously developed for Java. To implement renaming, we will follow the ideas presented in [12], extending the lookup attributes with inverse attributes that compute new accesses. By applying this technique to a challenging and fairly different language than Java, we expect to experimentally confirm the generality of this approach to renaming. Further work includes additional refactoring support, and including such support in a toolkit for building IDEs for new and extended languages.

Acknowledgments

We are grateful to our master's students Jesper Mattsson and Philip Nilsson for their implementation work on the JModelica IDE.

References

[1] The Modelica Association, 2009. http://www.modelica.org.

[2] J. Åkesson, T. Ekman, and G. Hedin. Implementation of a modelica compiler using jastadd attribute grammars. *Science of Computer Programming*, July 2009. doi:10.1016/j.scico.2009.07.003.

[3] T. Ekman and G. Hedin. Modular name analysis for Java using JastAdd. In *Generative and Transformational Techniques in Software Engineering, International Summer School, GTTSE 2005*, volume 4143 of *LNCS*. Springer, 2006.

[4] T. Ekman and G. Hedin. The jastadd system - modular extensible compiler construction. *Science of Computer Programming*, 69(1-3):14–26, 2007.

[5] T. Ekman and G. Hedin. The Jastadd Extensible Java Compiler. In *OOPSLA 2007*, pages 1–18. ACM, 2007.

[6] G. Hedin. Context-Sensitive Editing in Orm. In K. S. et al., editor, *Proceedings of the Nordic Workshop on Programming Environment Research*, Tampere University of Technology. Software Systems Lab. TR 14., 1992.

[7] G. Hedin. Reference Attributed Grammars. In *Informatica (Slovenia)*, 24(3), pages 301–317, 2000.

[8] G. Hedin and E. Magnusson. JastAdd: an aspect-oriented compiler construction system. *Science of Computer Programming*, 47(1):37–58, 2003.

[9] O. L. Madsen, B. Magnusson, and B. Møller-Pedersen. Strong typing of object-oriented languages revisited. In *OOPSLA/ECOOP*, pages 140–150, 1990.

[10] E. Magnusson, T. Ekman, and G. Hedin. Demand-driven evaluation of collection attributes. *Automated Software Engineering*, 16(2):291–322, 2009.

[11] Modelon AB. JModelica Home Page, 2009. http://www.jmodelica.org.

[12] M. Schäfer, T. Ekman, and O. de Moor. Sound and Extensible Renaming for Java. In G. Kiczales, editor, *23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2008)*. ACM Press, 2008.