LUND UNIVERSITY

Implementation of a Graphical Modelica Editor with Preserved Source Code Formatting

Sten, Jon; Mattsson, Tobias; Bergdahl, Tove; Mattsson, Jesper; Åkesson, Johan

Published in:
Proceedings of the 9th International Modelica Conference

DOI:
10.3384/ecp12076375

2012

Link to publication

Citation for published version (APA):
Sten, J., Mattsson, T., Bergdahl, T., Mattsson, J., & Åkesson, J. (2012). Implementation of a Graphical Modelica Editor with Preserved Source Code Formatting. In *Proceedings of the 9th International Modelica Conference* (pp. 375-384). The Modelica Association. https://doi.org/10.3384/ecp12076375

Total number of authors:
5

General rights
Unless other specific re-use rights are stated the following general rights apply:
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.
• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal

Read more about Creative commons licenses: https://creativecommons.org/licenses/

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

LUND UNIVERSITY

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

Download date: 12. Oct. 2024

Implementation of a Graphical Modelica Editor with Preserved Source Code Formatting

Tobias A. Mattsson[a] Jon Sten[a] Tove Bergdahl[c] Jesper Mattsson[c] Johan Åkesson[b,c]

[a]Department of Computer Science, Lund University, Sweden
[b]Department of Automatic Control, Lund University, Sweden
[c]Modelon AB, Sweden

Abstract

When an Integrated Development Environment (IDE) is developed, the support for multiple views of the same document is often essential. An example of this is Modelica models, where it should be possible to view and edit the same model in both its textual and graphical representation.

One implementation of Modelica is the open source platform JModelica.org. It contains the Eclipse-based JModelica.org IDE, providing a text editor for Modelica code based on the Eclipse platform.

In this paper, we present an implementation of a graphical editor for the JModelica.org IDE. Several challenges arising when implementing a graphical editor for Modelica models are discussed. Amongst others, the difficulties in rendering Modelica diagrams and how to interact with existing frameworks in Eclipse are covered. Also, a method for preserving the formatting of a modified source code file is presented, which is essential when the model is altered in the graphical editor.

The presented implementation is compared to other open source software (OSS) implementations of Modelica editors.

Keywords: AST; JModelica.org; Eclipse; GEF; Graphical Editing; Icon Rendering; Preserved File Formatting; Pretty Printing

1 Introduction

Simulation and optimization of dynamic systems is becoming a standard tool in several industrial branches. The trend is mainly driven by the demand for decreased product time to market and shortening the development time, by substituting system prototyping for simulation. Modelica is one of many domain specific languages developed with the goal to meet the demand of such model-based design languages.

One implementation of the Modelica language is the JModelica.org platform [1]. It contains a Modelica compiler as well as an Integrated Development Environment (IDE) for Modelica code. Currently, a comprehensive text editor for editing Modelica source code is available in the JModelica.org IDE [2], allowing the developer to define new models based on equations and existing models. The JModelica.org IDE is implemented using the Eclipse framework[1] which is a modular, extensible application framework for IDEs.

In this paper, we present an implementation of a graphical editor for Modelica in the JModelica.org platform which will complement the textual editor, already available in the JModelica.org IDE. The editor is implemented as an Eclipse plugin using the Graphical Editing Framework (GEF)[2] which is a framework for creating graphical editors, developed for the Eclipse platform. The graphical editor communicates and modifies Modelica models through an abstract syntax tree (AST). It also features preserved file formatting in the JModelica.org IDE. The work presented in this paper is the result of two master's theses [3, 4], conducted at Modelon AB.

This paper is outlined as follows. In Section 2, a brief background of the JModelica.org platform is given and the compiler construction framework JastAdd [5] is introduced. The Eclipse project and the Graphical Editing Framework (GEF) are also introduced in this section. In Section 3, a comparison to similar OSS tools is presented. The implementation of the graphical editor is discussed in Section 4 and Section 5 summarizes this paper.

[1] `http://eclipse.org`
[2] `http://eclipse.org/gef`

2 Background

2.1 JModelica.org

JModelica.org is an open source project for optimization and simulation of complex dynamic systems. The JModelica.org platform includes compilers for Modelica and the Modelica language extension Optimica [6], as well as an integration to the simulation package Assimulo [7]. An interface to the compilers and simulation and optimization algorithms is available in Python, which enables scripting of the typical modeling and optimization activities.

Also part of the JModelica.org platform is an IDE for Modelica. The JModelica.org IDE is implemented as a plugin in Eclipse using the JModelica.org compilers and the JastAdd framework. The IDE provides textual editing support such as syntax highlighting, code folding, code outline, brace matching and error checking of models.

2.2 Eclipse

The Eclipse Foundation is an open source community whose aim is to produce open development platforms with comprehensive extension frameworks[3]. The IDE is heavily modularized so that it is possible to add, remove and extend functionality with a small amount of code and without altering any core source files. The modularization also makes it possible to create different bundles, including different editors and views. For instance, there is the Eclipse Software Development Kit (SDK) that includes a comprehensive Java Development Tool (JDT) for Java development and also the C++ Development Tool that is an IDE for C and C++. These are two different development environments with different functionality, yet they still use the same base IDE and base functionality.

2.3 GEF

The Graphical Editing Framework (GEF) is one of the most popular frameworks for graphical editing in Eclipse and it is also the one used for the editor in this paper[4]. GEF is a rather complicated system with many design patterns and classes. When developing graphical editors using GEF, the developer has to define two types of classes, *EditParts* and *EditPolicies*.

EditParts is the most basic part of GEF. These classes join the document model with the view. There is usually a one to one representation between document model nodes and EditPart classes. The view is represented by figures. Normally, figures also map one to one with EditParts, see Figure 1.

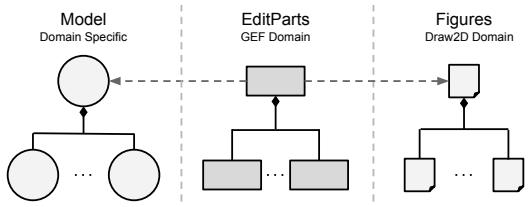


Figure 1: View of the document model, EditPart and figure tree and their linkage.

EditPolicies handle the interaction with the user, the Eclipse framework and the underlying model. For example, the graphical editor specifies an EditPolicy that determines what should happen when the user tries to move a component. If the move is valid, it will create a move command that alters the model component definition.

An EditPolicy specification usually only handles a single task or a group of related tasks. Using this pattern means that the user interaction is separated from the EditParts. Instead, the interaction is handled by EditPolicies installed on EditParts. This enables different EditParts with similar behavior to use the same EditPolicies, which reduces code complexity. It is also convenient for the developer, since it allows for gradually extending the functionality with new features.

2.4 JastAdd

JastAdd[5] is an open source meta-compilation system that is used for compiler generation and other programs that have the need to analyze code. It provides means to define attribute grammars [8], and introduces the possibility to use *aspect*-oriented programming (AOP) when constructing a compiler. With aspects, the source files describe a certain behavior or functionality, rather than objects in object-oriented programming (OOP). In other words, the behavior for several different objects may be defined in the same aspect.

JastAdd code is organized in abstract grammar files and aspect files. These source files are collected and the functionality from the aspects is woven into Java files before they are finally compiled.

The JastAdd project provides a framework for supporting IDEs based on Eclipse [2]. It consists of a generic IDE plugin with supporting classes and default

[3] `http://eclipse.org`

[4] `http://eclipse.org/gef`

[5] `http://jastadd.org`

aspects for attributes. The attributes provide common services such as code folding and code outline. The main parts of the generic IDE plugin are the *builder* and the *registry*. When the Eclipse framework needs a build, it triggers the builder. The builder then delegates the work to a compiler. When the compiler is done, it provides an AST for files or projects. These ASTs are cached in the registry.

2.5 Graphical Annotations

In this paper, Modelica annotations, or more specifically graphical annotations, will play an important part since they are used for representing a model and its components graphically. A graphical editor uses the information in the graphical annotations when rendering icons and diagram. The editor also modifies annotations when the user makes changes in the graphical editor.

Listing 1: Code example of graphical annotations in the three different locations permitted.

```
model LowPass
  ...
  Analog.Basic.Resistor R1
  annotation(Placement(transformation(
    extent={{-25, -25},{25, 25}},
    origin={-25, 50}
  )));
  ...
equation
  ...
  connect(R1.n, p2)
  annotation(Line(points=...));
  ...
  annotation(
    Icon(
      coordinateSystem(extent=...),
      graphics={...}
    )
  );
end LowPass;
```

There are three locations where graphical annotations may appear:

(a) Directly after a component definition. If specified, it will define how that component should be rendered, size and origin.

(b) Directly after a connect statement. If specified, it will define how the connection should be rendered, line points and its color.

(c) At the end of a model definition. If specified, it will define how the icon and diagram of the model should be drawn.

Examples of the three different locations are illustrated in Listing 1.

3 Related Work

There are several approaches to formatting preservation and graphical Modelica editing. In this section, a comparison will be given to some of the popular open source alternatives.

3.1 Formatting Preservation

OpenModelica[6] is another open source initiative based on Modelica. The OpenModelica environment also has procedures for preserving formatting. Peter Fritzson et al. describes these procedures as an initiative to preserve comments and indentation when refactoring Modelica code [9].

OpenModelica stores the text representation of the code in a separate tree, which has the same structure as the original AST. In this way, they are able to avoid cluttering the AST with text positions. The text representation is created piece-wise when needed. The nodes in this separate tree stores the text positions and have a one-to-one mapping with the nodes in the AST.

While the solution for preserving formatting presented in this paper also aims to do most of the work when it is actually needed, it does more work in the parsing process. The position of the nodes and the actual formatting text and type is extracted during parsing. This data is then associated to the source AST. This makes the AST slightly more memory consuming at first, compared to the OpenModelica solution.

The most significant difference between the two solutions is that in JModelica.org, the formatting always resides in the source AST while OpenModelica stores it in a separate tree. Although this makes the source AST in JModelica.org more verbose, the advantage is that modifying the source AST does not require any synchronization with a second tree.

Maartje de Jonge and Eelco Visser have also presented an algorithm for preserving the original layout of source code when modifying an AST [10]. Their algorithm relies on text reconstruction and origin tracking. The algorithm stores a reference to the leftmost and rightmost token in the stream for each node in the AST, which in turn holds the corresponding start and end offset. When the AST is to be modified, the nodes and their positions in the new tree are traced back to

[6] http://openmodelica.org

their origin. The text can then be reconstructed from this origin.

The algorithm also comes with an intelligent heuristic for associating comments with the correct node. Cases such as block comments, comments before and after a line of code, inside comma separated lists, code removed by commenting and multi-line comments beside multiple statements are discussed. Suggestions how to handle most of these cases are also described.

The solution presented in this paper is less involved than the algorithm by de Jonge and Visser, but still covers the realistic cases threated in this paper. All comments in the source code that are located on the right-hand side of a node, but before a line break or the next node, are associated with that node. Any comments that follow are considered to belong to the next node, and so forth. It is important to remember that comments are meant for people, not machines, to read and interpret. Thus, there are no predefined rules for how to relate comments to code and it is practically impossible to perfect such an algorithm.

As the JModelica.org IDE is an Eclipse plugin it is worth mentioning how the Eclipse Java Development Tools (JDT) handles changing the AST. The Eclipse JDT has an API for refactoring code using the AST [11]. When AST nodes are added, removed or replaced in JDT, these operations are translated into text edits which can then be applied to the original source. This is a very different approach than ours, as this means that the original AST is never touched by JDT. Instead the source text is edited and the AST is then updated from that source. In JModelica.org, such an approach would require a total recompilation of the code with every AST modification as there is currently no way to incrementally compile the source code. This would thus not be an adequate solution, as it would most likely make the editor very slow.

3.2 Graphical Modelica Editors

OMEdit is an editing front-end to the OpenModelica compiler. It contains tools for model creation, diagram editing, icon editing, simulation, plotting, documentation view and text editing mode. The editor was developed as part of a thesis by Asghar, Syed Adeel and Tariq, Sonia in 2010 [12, 13].

OMEdit uses the CORBA interface to communicate with OpenModelica. CORBA is supplied by OpenModelica and allows for interaction between an application and its AST. OMEdit uses it to retrieve and store information such as: model structure, annotation, documentation, simulation and graph plotting.

OMEdit is mainly developed in C++ and relies on *QT*[7] for graphical UI handling and rendering of graphical primitives. QT is an cross platform framework that allows the developer to rapidly develop Graphical User Interface (GUI) based applications that work on multiple platforms. It also has support for multiple target languages like C++ and JavaScript.

Compared to the graphical editor presented in this paper, OMEdit is more comprehensive. It defines a complete IDE with text editor, parameter view, simulation view and graphical editor.

There are some significant differences between the two graphical editors, besides the programming language. OMEdit uses QT to generate a GUI, as opposed to GEF that is used for the graphical editor in this paper. GEF and QT provide the same basic functionality but for different programming languages and platforms. However, QT has more extensive support for the graphical features that are specified in Modelica than GEF. QT also takes care of all transformation and rendering of graphical primitives.

Performance wise there are some small differences between the two graphical editors. When adding and removing components a noticeable lag is present in OMEdit editor while it happens instantly in JModelica.org editor. This is most likely a result of the CORBA interface and the fact that OMEdit does not operate directly on the AST.

OMEdit also lacks some basic features like an undo and redo stack. This is likely due to that support for this is missing in QT. This feature is something that was supported by GEF and is one of the essential features in the graphical editor described in this paper.

SimForge[8] is another open source toolkit that is abased on OpenModelica. It offers similar features as the JModelica.org IDE and OMEdit. It has a text editor as well as a graphical editor that allows for both diagram and icon editing. Additionally it has a parameter editor that allows for modification of component values. The SimForge project has been inactive for some time and there is also a lack of information about the implementation and the frameworks used. Therefore, no thorough comparison is given in this paper.

[7] `http://qt.nokia.com`

[8] `http://trac.ws.dei.polimi.it/simforge/`

4 Implementation

4.1 Compiler Architecture

The JModelica.org compiler is divided into two parts, front-end and back-end. The front-end is responsible for parsing, AST building, error checking and flattening of Modelica models. The front-end builds three ASTs, the source AST, the instance AST and the flat AST.

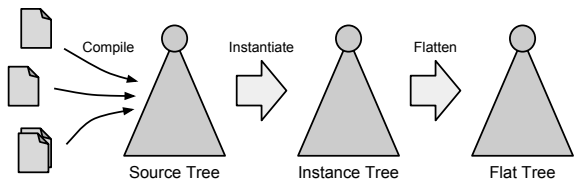


Figure 2: The three ASTs and the different compilation stages.

Source AST is the first AST that is built. It is almost a direct translation of the Modelica model into a tree. The source AST is necessary when calculating the instance tree.

Instance AST is instantiated from the source AST. All component declarations has been resolved and expanded with the contents of their classes.

Flat AST is the flattened version of the model. It is reduced from the instance tree and consists of a list of equations and variables.

A brief overview of the three ASTs can be seen in Figure 2.

4.2 Graphical Editor

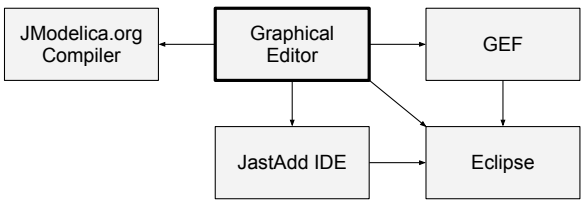


Figure 4: An overview of the interaction between the graphical editor and the other components.

An overview of the design and the different components that the graphical editor relies on is shown in Figure 4. The graphical editor communicates with the JModelica.org compiler to retrieve graphical annotations as well as Modelica class structure. All editing is saved back into the source and instance AST and the icon structure. The icon structure is an abstraction of the structure that is used when representing Modelica annotations in the source AST. It is built to resemble the structure of a graphical annotation as it is defined in the Modelica specification. GEF is an obvious component that the editor relies heavily on. GEF helps the editor with synchronization between the EditParts and icon structure in the compiler. It is also responsible for interaction with the Eclipse framework and low level rendering of graphics in the editor. The graphical editor also has some direct interaction with Eclipse, such as the ability to open a component for modification of parameters on sub components. The JastAdd IDE framework is used as an interface to the JModelica.org compiler when compiling classes.

4.2.1 EditParts

When constructing a GEF editor, it is important to make a good design between EditParts and the underlying document model. Normally, there is one EditPart class for each document model node. The graphical editor presented in this paper is no exception and uses one EditPart class for each icon structure node. For example, the icon structure node *Rectangle* is represented by a *RectangleEditPart*. The *RectangleEditPart* handles all the rendering and interaction with the graphical user interface (GUI). By using EditPolicies, it is possible to alter the behavior and control what happens when the rectangle is moved or resized. Similarly, there will be one EditPart class for each node type in the icon structure, specifying the behavior of that node.

4.2.2 Rendering

Once an EditPart has been produced from the icon structure, it creates an appropriate figure and populates it with the correct attributes from the icon structure. For example, the rectangle mentioned in Section 4.2.1 will populate its figure with width, height and rotation. It will also set line color, line pattern and fill color.

It can sometimes be troublesome to render Modelica models in GEF. Modelica supports both rotation and scaling of graphics whilst GEF does not support rotation and has limited support for scaling. In the graphical editor this is solved by transforming the points that the graphical object consists of. The transformation is done using Euclidean transformations, that are built hierarchically over the component structure of the Modelica model.

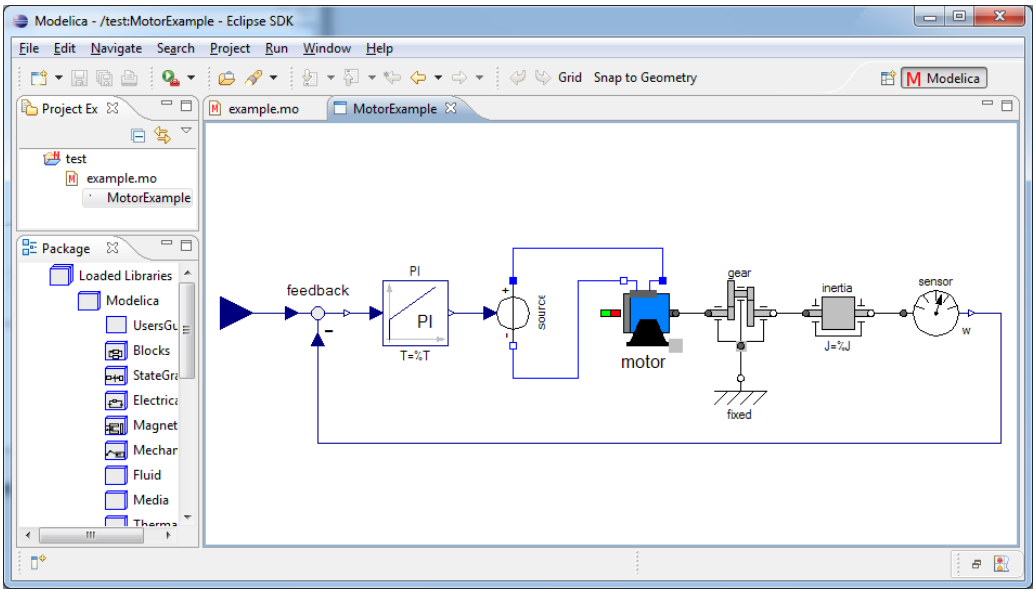
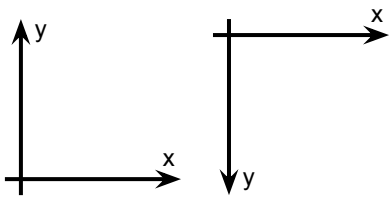


Figure 3: Screen shot of the graphical editor.



(a) Modelica (b) SWT

Figure 5: Difference between Modelica and GEF coordinate system with positive direction indicated by arrows on the axis.

Finally, when rendering models it is important to convert between the handedness of the coordinate systems used by Modelica and GEF respectively. Modelica uses a right-hand coordinate system, see Figure 5a while GEF uses left-hand coordinate system, see Figure 5b. If no consideration is taken to handedness the resulting image will be upside down. The solution is simple, once all transformations are done the image is flipped along the y-axis.

4.3 AST Communication

Changes made by the user in the graphical editor has to be propagated back into the icon structure and underlying source AST. These propagations can be divided into two categories, annotation editing and structural editing.

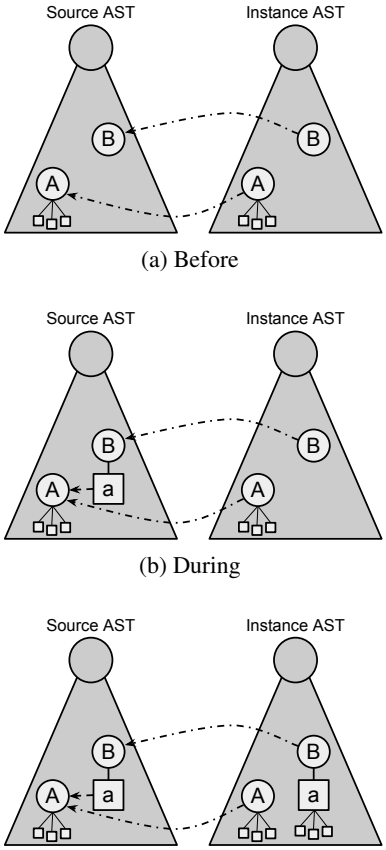
4.3.1 Annotation Editing

Some graphical changes, such as moving or resizing components, are localized and only affects annotations in the code. Most of the operations performed while graphically editing a model falls into this category. Since this kind of change only affects the source AST, it is simple to perform.

4.3.2 Structural Editing

Structural editing is a more complicated type of editing operation which is performed on the AST. It occurs when a *component* or a *connection* is added to or removed from the model. The main challenge is that both the source and instance tree must be updated consistently. In the current implementation, the component or connection is first added to the source tree. The instance tree then instantiates a new component or connection from the source node, resulting in a consistent result for the currently opened model, see Figure 6. Removing a component is performed in the reverse order as opposed to adding. First, the component or connection is removed from the instance tree and then in the source tree.

There are, however, side effects that are not handled. If the edited model is used as a component in another model and that other model is also open, the latest changes will not appear until that model is reloaded. A possible solution is to sense when a structural edit has occurred and in that case reload the editor. Another solution is to propagate any changes in a model to all instances of the same model.



(a) Before

(b) During

(c) After

Figure 6: The steps taken when adding a component.

4.4 Preserved Formatting

Consider the graphical editor as currently described in this paper. When the editor changes the model, it does so by modifying the source AST. Eventually these changes need to be displayed in a source code editor or saved to a file. The source AST would then have to be printed back to text format. All information that is significant to the compiler and the graphical editor resides in the AST. It does not, however, traditionally carry any knowledge about how the code was originally formatted. Information that is valuable for the developer, such as indentation and comments would effectively be lost if the AST was simply pretty printed. Somewhere along the way, the information about the original formatting needs to be gathered and stored in the AST so that the original source code can be reprinted.

4.4.1 Scanner and Parser

The *scanner* is the part of the compiler that finds patterns in the code and converts them to a series of tokens given to the *parser*. Those tokens are then converted to a source AST by the parser using the grammar of the language. The issue about preserving formatting, as described in the paragraph above, is solved by letting the scanner add spaces, line breaks and comments to a data structure. Most parentheses are implicit by the Modelica language, but expressions need special care regarding this. The developer might surround expressions by parentheses to explicitly mark precedence. This could, for the sake of readability, be done even when it would already be implicit. The parser collects these parentheses and stores them in respective expression.

When the parsing is finished, a reference to the data structure is added to the nodes in the AST. Later, when the AST is about to be presented in text format, for example, when it should be saved to a file, the information in the data structure is propagated downwards in the AST. At this stage each node gets the formatting related to the node, as the data structure is emptied. After this, the AST is finally printed in text format, where every node has its formatting preserved.

4.4.2 Reading Formatting

As has been mentioned earlier, formatting such as indentation and comments are put in a data structure by the scanner. Some, but not all parentheses should also be collected. The parser has, unlike the scanner, the syntactic information to distinguish which parentheses are significant. That is, which parentheses belong to expressions and which do not. However, the parser has no access to the formatting data structure, so it stores the parentheses directly in the expressions.

When the data structure has been populated by the scanner, it contains a list of *scanned formatting items*. A *formatting item* is an object that contains some basic part of the formatting. It contains the actual string data that should be output when the AST is printed. A formatting item also has information about what type it is, for example *line break* or *comment*. When a formatting item has been scanned it is called a *scanned formatting item*. A scanned formatting item holds the same information as a regular formatting item, but with some information about its origin added. This makes it possible to find its original line and column in the source code. Table 1 shows some typical scanned formatting items.

Before the formatting items in the data structure are used by the AST, some of them are merged. The ones that are adjacent to each other are merged into new, larger formatting items of mixed type. As an example, the two last items in Table 1 would be merged into a

Table 1: The data in some scanned formatting items.

Scanned Formatting Item			
		Formatting Item	
Start	End	Type	Data
(1, 6)	(1, 6)	WHITE_SPACE	” ”
(1, 19)	(1, 19)	LINE_BREAK	”\n”
(2, 1)	(2, 4)	WHITE_SPACE	” ”
(2, 9)	(2, 9)	WHITE_SPACE	” ”
(2, 13)	(2, 13)	WHITE_SPACE	” ”
(2, 14)	(2, 29)	COMMENT	”// Text\n”

mixed formatting item. This way each AST node only needs to be associated with one formatting item that can be seen as a prefix, or left-hand side, and one as suffix, right-hand side.

4.4.3 Storing Formatting in the AST

The formatting items are not added to the AST nodes during parsing. Instead they are added during an extra pass when the method for formatted print is called. This approach naturally comes with its advantages and drawbacks.

Approach One of the main reasons for the chosen approach, is to keep the parser as separated from the implementation of this feature as possible. A parser can be complicated enough to begin with. Furthermore, if it turns out that there are any special cases that need to be taken care of, or if parts of the parser need to be rewritten it could become cumbersome and expensive to implement and maintain the code.

Initially, another reason for this approach was that some AST nodes might be *rewritten* and removed when the AST is being accessed. This means that they are replaced with other nodes to make the AST more suitable for semantic analysis [14]. Their formatting would then effectively also disappear. However, this issue is not completely solved by adding the formatting to the AST at a later stage. There are some AST nodes in JModelica.org that are deleted during rewrites that both can come from an explicit keyword or be implied by the Modelica language specification. As an example, members of a Modelica class can be specified to have public or protected visibility[9], but their default visibility type is public [15]. A public visibility clause is thus an AST node that can come both from a keyword *or* the language specification. This clause is removed during a rewrite, and its child nodes get their visibility by assigning them visibility type child nodes. The fact whether the keyword appeared in the source code or not, and if so then where it appeared, still needs to be stored.

The solution presented in this paper comes with a drawback. It is more intense for the CPU to add the formatting to the AST on-demand, rather than in the parser. Firstly, the data structure needs some preparations. Secondly, all AST nodes need to be associated with their corresponding formatting items. This means that the line and column numbers for AST nodes and formatting items need to be compared. It is worth noting, though, that the result from this pass can be cached. This means that consecutive prints of the AST do not need this pass and no more clock cycles are used for this.

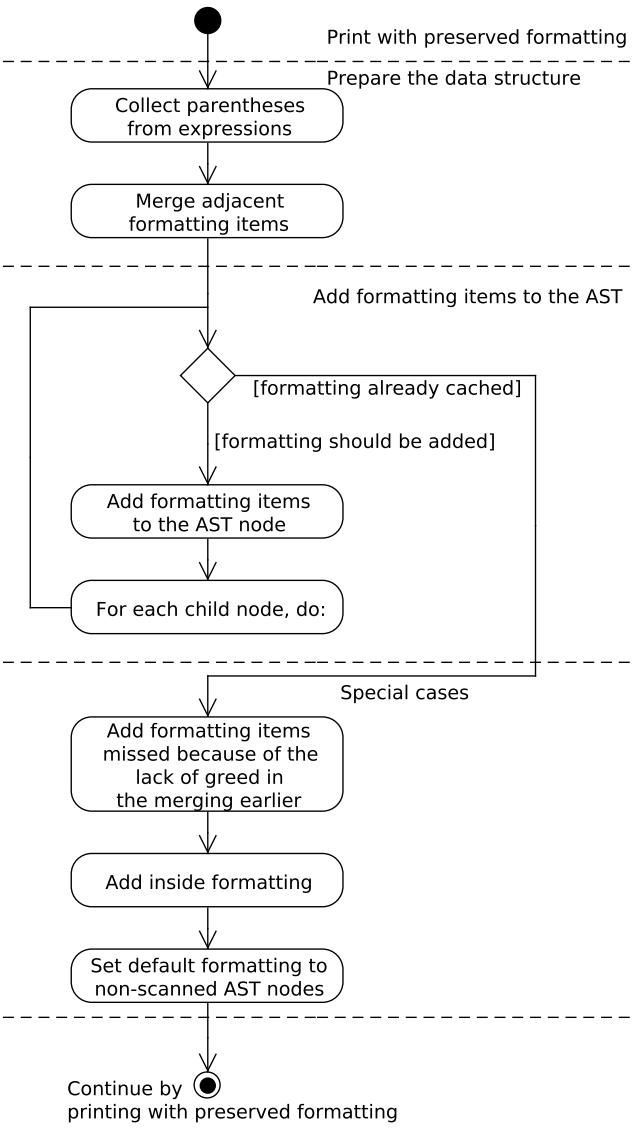


Figure 7: The process for storing formatting in the AST.

[9]Using the keywords `public` and `protected` respectively.

Preparations Figure 7 shows the main steps of how the formatting is added to the AST. The first step is to prepare the formatting information from the parsing to be propagated downwards in the tree. The parentheses that were collected to the expressions during parsing are added to the data structure. After that, the formatting items that are adjacent are merged as mentioned in Section 4.4.2. To have a more logical division between the formatting items that should be considered prefix or suffix to an AST node, this algorithm is not *greedy*. This lack of greed means that merged formatting items that span over multiple lines are split into two items on the first line break, instead of always being merged into one. An example of this can be seen in Figure 8. After these preparations, the AST nodes can use the data structure to get their formatting items.

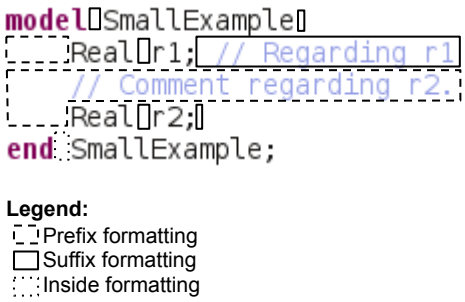


Figure 8: A screenshot highlighting the different formatting items with boxes. Note that the comments regarding `r1` and `r2` are not merged into one single mixed item.

Propagation During the propagation, the nodes in the source AST check so that they do not already have a cached result. This happens if the AST already has been reprinted earlier. If they do not have any cached formatting, their formatting is calculated. This is done by going through the data structure filled with formatting items and checking whether they are adjacent to the current AST node, that is, whether their end line and column match the AST node's starting position and vice versa. In this way, each AST node gets two formatting items, one on its left-hand side (prefix) and one on its right-hand side (suffix).

Special Cases If there for some reason are any formatting items left in the data structure when the propagation is done, these still need to find their place in the AST. This can happen, because the merging of adjacent formatting items sometimes generates two formatting items instead of one as described earlier. If the source code in a file ends with multiple line breaks, only the first one would be added without an extra step.

There are also some AST nodes that contain formatting information inside of them. These nodes usually contain whitespaces at places where they are *atomic*. They are atomic in the sense that they have no child nodes that can use the whitespace as prefix or suffix formatting. These final formatting items left in the data structure are also added.

Finally, a default formatting is set to AST nodes that have been added through another way than during the parsing of the code. Currently, this means AST nodes that have been added by the graphical editor. When this is done, the rest of the implementation is more or less a traditional pretty printer, which of course also prints the formatting information in the AST nodes.

5 Summary and Conclusions

In this paper, an approach for implementing a graphical editor for Modelica built upon the Eclipse framework using GEF, was presented. How GEF can be used to make a clear, yet extensible design for the graphical editor has been discussed. Some of the common pitfalls when integrating systems that describe the same information in different ways, in this case integrating a graphical editor into an existing source code editor, were discussed.

This paper also describes a way to store formatting from an original source in the AST. In the proposed solution, the formatting information is added to the source AST after parsing. Then in a later pass, the information is associated with its corresponding node. Finally, the modified AST can be reprinted with preserved formatting.

The graphical editor in JModelica.org supports basic model editing such as adding, removing and connecting components. Common graphical editing features such as rotation of components and grid snapping are also available. Future development include a parameter dialog for modifying of parameters and improved graphical editing support such as manhattanized connections.

References

[1] J. Åkesson, K-E. Årzén, M. Gäfvert, T. Bergdahl, H. Tummescheit. Modeling and optimization with Optimica and JModelica.org—languages and tools for solving large-scale dynamic optimization problem. *Computers and Chemical Engineering*, 34(11):1737–1749, November 2010. Doi:10.1016/j.compchemeng.2009.11.011.

[2] J. Mattsson, The JModelica IDE: Developing an IDE by Reusing a JastAdd Compiler, Master's thesis, Department of Computer Science, Lund University, Sweden, 2009.

[3] J. Sten, Graphical Editing in JModelica.org, Master's thesis, Department of Computer Science, Lund University, Sweden, 2012.

[4] T. Mattsson, AST-driven Editor, Master's thesis, Department of Computer Science, Lund University, Sweden, 2012.

[5] G. Hedin, E. Magnusson, JastAdd: an aspect-oriented compiler construction system, *Science of Computer Programming* 47 (1) (2003) 37–58. doi:http://dx.doi.org/10.1016/S0167-6423(02)00109-0.

[6] J. Åkesson. Optimica—an extension of modelica supporting dynamic optimization. In *In 6th International Modelica Conference 2008*. Modelica Association, March 2008.

[7] C. Andersson. A new Python-based class for simulation of complex hybrid DAEs and its integration in JModelica.org. Master's thesis, Department of Mathematics, Lund University, Sweden, 2010.

[8] G. Hedin, "An Introductory Tutorial on JastADD Attribute Grammars," in *Generative and Transformational Techniques in Software Engineering III*, ser. Lecture Notes in Computer Science, vol. 6491. Springer-Verlag Berlin Heidelberg, 2011, pp. 166–200.

[9] P. Fritzson, A. Pop, K. Norling, M. Blom, Comment- and Indentation Preserving Refactoring and Unparsing for Modelica. In *6th International Modelica Conference 2008*, pp. 657–665. Modelica Association, March 2008.

[10] M. de Jonge, E. Visser, An Algorithm for Layout Preservation in Refactoring Transformations. Software Language Engineering, pp. 40–59, Springer, 2012.

[11] Eclipse Documentation: ASTRewrite, `http://help.eclipse.org/indigo/topic/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/dom/rewrite/ASTRewrite.html`, 2012.

[12] S. Asghar, S. Tariq, Design and Implementation of a User Friendly OpenModelica Graphical Connection Editor, Master's thesis, Department of Computer and Information Science, Linköping University, Sweden, 2010.

[13] S. Asghar, S. Tariq, M. Torabzadeh-Tari, P. Fritzson, A. Pop, M. Sjölund, P. Vasaiely, W. Schamai, An Open Source Modelica Graphic Editor Integrated with Electronic Notebooks and Interactive Simulation. In *8th International Modelica Conference 2011*, pp. 739–747. Modelica Association, March 2011.

[14] T. Ekman, G. Hedin: Rewritable Reference Attributed Grammars. ECOOP 2004. LNCS, vol. 3086, pp. 147–171. Springer, Heidelberg (2004).

[15] Modelica Association, Modelica - Language Specification 3.2 rev 1, 2012. `http://modelica.org/documents/ModelicaSpec32Revision1.pdf`.