Rewriting JGrafchart with Rewritable Reference Attribute Grammars

Theorin, Alfred; Årzén, Karl-Erik; Johnsson, Charlotta

2012

Rewriting JGrafchart with Rewritable Reference Attribute Grammars

Alfred Theorin, Karl-Erik Årzén, and Charlotta Johnsson

Department of Automatic Control, Lund, Sweden
{alfred.theorin,karlerik,charlotta.johnsson}@control.lth.se

Abstract. Grafchart is a graphical programming language for sequential control applications. It exists in two versions: the basic version (BV) and the high-level version (HLV). The currently used Grafchart tool, JGrafchart, only supports BV. To enable further research on HLV, it must be supported by JGrafchart. Since HLV is a superset of BV it is desirable to add it as an extension to the current implementation of BV. Rewritable Reference Attribute Grammars (ReRAGs) have been successfully used to implement several other extensible compilers. Grafchart consists of one graphical and two textual sub-languages. This paper focuses on making the two textual sub-languages extensible by rewriting them using ReRAGs. The *sup* notation is added as an extension to the ReRAGs implementation to confirm extensibility.

1 Introduction

Grafchart is a graphical programming language for sequential control applications that has been developed at Lund University. It is based on Grafcet/Sequential Function Charts (SFC) and is suitable both for applications on the local and on the supervisory level [9], as well as on both the Business and the Manufacturing Execution System (MES) layers, and in all phases from process-planning to execution [5]. Furthermore, Grafchart is easy to use and understand and it has potential for formal descriptions, validation, and analysis [7].

Grafchart exists in two versions: the basic version (BV) and the high-level version (HLV). BV is based on Grafcet and has extended syntax and facilitates additional abstractions. HLV contains additional high-level programming language constructs as well as features inspired by high-level Petri nets. These constructs make HLV very expressive and well suited for a wide variety of control applications, e.g. it has been used to create convenient batch control applications which otherwise tend to become very complicated [7].

Toolboxes for both BV and HLV have been implemented in G2 [4]. BV is also implemented as a freeware, standalone Java program, JGrafchart, which supports editing, compilation, and interactive interpreted execution. For various reasons, the G2 toolboxes are no longer used. Instead JGrafchart is the tool of choice and is used both in education and research.

2 The Problem

To enable further research on HLV, JGrafchart must first support the already existing HLV features. It is desirable to retain a pure BV as well. Since HLV is a superset of BV it is desirable to re-use the BV implementation as a base for the HLV implementation to avoid code duplication and to improve maintainability. This imposes an extensibility requirement on both the editor and the compilers.

3 ReRAGs

Rewritable Reference Attribute Grammars (ReRAGs) extend the attribute grammars introduced by Donald Knuth [8], adding several new concepts such as reference, collection, and parametrized attributes and node rewrites. ReRAGs are implemented in the open source compiler compiler system JastAdd [6].

The main difference between attribute grammars and traditional compiler techniques is that attribute grammars are declarative while traditional compilers are imperative. Instead of explicit abstract syntax tree (AST) traversal, the semantics are specified using equations. One big advantage with declarative programming is that one does not have to consider the order in which the properties are calculated since this is handled automatically by the evaluation framework.

JastAdd has been used to successfully implement other extensible compilers for a wide variety of purposes, e.g. the JastAdd Java Compiler (JastAddJ) that is written as a Java 1.4 compiler with a Java 1.5 extension [3], Control Module Language with object oriented extension [2], and the Optimica extension to Modelica [1]. It thus appeared to be a suitable candidate for making the JGrafchart compilers extensible.

4 Grafchart

Graphical languages are popular in the automation community, e.g. three of the five languages in the PLC standard IEC 61131-3 are graphical. The advantages of graphical programming languages are simplicity and declarativeness and they often allow programming in the same way as people model problems.

Interactive, graphical execution makes it easy for the operator, i.e. the intended typical user, to monitor the current execution state. To make it even more appealing to the operators it is also possible to create more intuitive operator interfaces using the many graphical elements in JGrafchart. Typically the operator interface resembles the controlled process, e.g. a set of tanks connected by piping, with indicators and interactive elements added, see Figure 1.

Grafchart is based on the graphical syntax of SFC, one of the graphical languages of IEC 61131-1, which is well-accepted by the industry today. Grafchart has also incorporated ideas from statecharts, high-level Petri nets, and ordinary object oriented programming languages to extend the rather low-level SFC language with constructs for hierarchical structuring and exception handling. This makes it possible to create large, well structured, and maintainable applications with support for formal analysis.

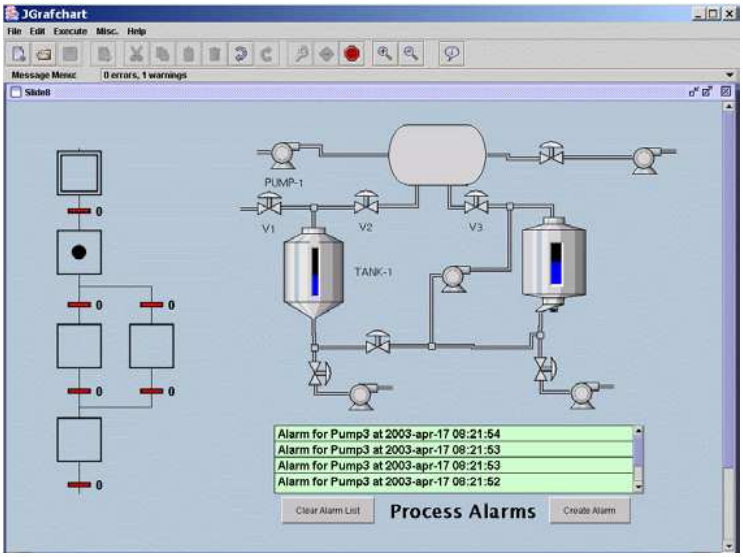

Fig. 1. A dummy control application and operator interface created in JGrafchart.

4.1 Syntax

Grafchart consists of steps, representing states, and transitions, representing the change of state. Steps and transitions are connected by arcs. The current state is indicated by tokens in the steps and a step is active if it contains tokens. Associated with the steps are actions that are executed at certain times, e.g. when the step is activated (S action) or deactivated (X action). The N action associates setting/resetting of a boolean variable with the step activation/deactivation. To each transition a boolean condition is associated. A transition is enabled when all its previous steps are active. An enabled transition fires if its condition is true, meaning that its previous steps are deactivated and its next steps are activated.

The hierarchical constructs which can be used to group sub-sequences are the macro step and the procedure. A macro step is a step containing a sub-sequence while the procedure is not a step but is instead re-usable and can be called from procedure steps and process steps. Procedure steps are similar to function calls in ordinary programming languages. Process steps on the other hand spawn a separate execution thread for the procedure execution upon each activation and do not have to wait for the called procedure to reach its final state before proceeding to the next step.

In BV there is essentially only one token while in HLV it is possible to have several tokens, and to spawn and consume tokens dynamically. It is also possible for the tokens in HLV to contain data (compare to colored Petri nets) that may be used in the actions and conditions.

4.2 Example Application

The application in Figure 2 implements a controller for a batch tank that is filled until `full`, then emptied until `empty`. This sequence is repeated, and each time the filling is initiated the `cycles` counter is incremented. In the current execution state the 4th filling has just been initiated.

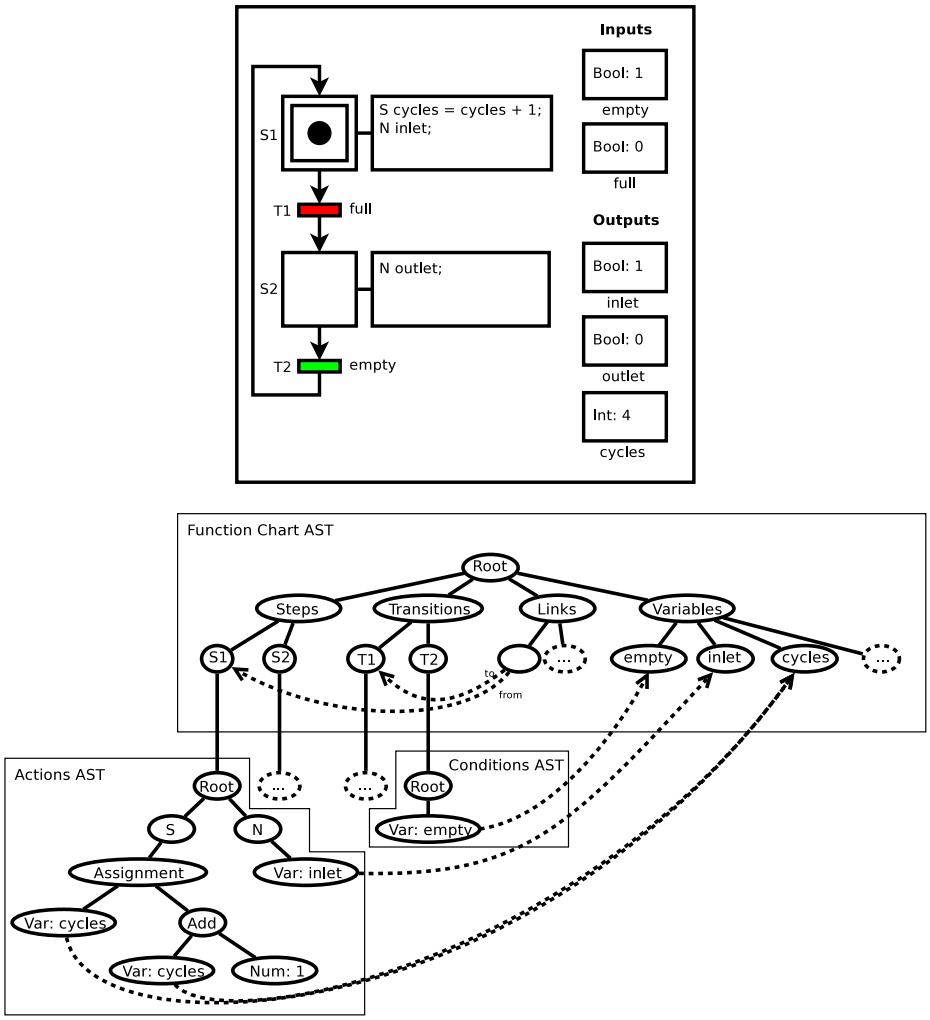

Fig. 2. A Grafchart batch tank control application and its AST with references.

4.3 Inputs and Outputs

JGrafchart contains a wide variety of means of connecting various I/O to interact with the external environment. One way is with custom Java implementations of analog and digital I/O. More general I/O can be implemented using Socket I/O. Devices Profile for Web Services (DPWS) devices can also be used without a priori knowledge about their capabilities and without any custom code [10].

This means that JGrafchart can be connected to practically any external environment, in most cases with only a small or moderate effort.

4.4 Language

The Grafchart language consists of three sub-languages: The function chart language (graphical), the actions language (textual), and the conditions language (textual). The function chart language contains the steps, transitions, and other graphical elements such as variables, I/O, and procedures. The actions language is used for the actions of the steps. The conditions language is used for the transitions specifying when to move from one step to another. It consists of the expressions subset of the actions language with flank and event detection added.

It is only the extensibility of the textual sub-languages of Grafchart that is considered in this paper.

5 Refactoring

The compilers for the actions and conditions languages were previously written using traditional compiler construction techniques and tools. Using JavaCC the scanner and parser specifications were used to generate classes for scanning, parsing, and building the AST. The language semantics were then added by inserting code into the generated files. Interpreter code for execution was also added to the same files. This approach has the following drawbacks:

- The semantics code, the interpreter code, and the generated code are intermixed making it hard to distinguish the different parts.
- The functionalities are hard to overview since they are split up in all contributing Java classes.
- The semantics are written using imperative programming and are thus inherently hard to extend.

Trying to create an extension under these circumstances would be error-prone. Instead the following strategy for refactoring was used:

1. Separate the hand-written code from the generated code.
2. Split the hand-written code into logical modules based on functionality.
3. Simplify the semantics analysis by using ReRAGs.

These steps were applied to both the actions and the conditions language implementation in turn, for convenience the conditions language was considered first as it is almost a subset of the actions language.

5.1 Step 1: Separation

Separating the hand-written from the generated code was straightforward. The parser was re-generated into an empty directory and then compared to the current code. The code that was not present in the re-generated files was considered hand-written and moved into one single, large JastAdd module. For this to work in JastAdd, the AST node types also had to be specified. As a starting point the AST specification was simply a listing of all the AST node types. Later it was rewritten to make the AST structure explicit.

5.2 Step 2: Split Into Modules

The modules chosen for both language implementations were Compiler, Interpreter and Utilities. The Compiler module handles the compilation of the AST, the Interpreter module handles interpreted execution of the compiled AST, and the Utilities module contains helper functions that are not specifically related to any other module.

A part of the actions and conditions language implementations were the built-in functions and methods such as `abs()`, `min()`, and `getWidth()`. These were compiled and executed separately by each language implementation in the old implementation. However, they do not fit in either module since information about them is required during compilation and their implementation is required during execution. Since most built-in functions and methods are available both in the actions and conditions languages it is also more appropriate to only have them implemented once. Therefore they were instead extracted to a separate package that is used by both language implementations during both compilation and execution.

5.3 Step 3: Simplification

After the split into modules, the Compiler modules were transformed piece by piece to ReRAG equations in parallel with creating JUnit tests for verification.

In the old implementation, a one pass traversal of the entire AST was performed. Compilation messages were sent directly to the editor during the traversal, and to know if the compilation was successful, a separate boolean variable was propagated upwards in the tree and returned by the root node.

The new implementation uses a collection attribute in the root node for the compilation messages. This means that the compilation success status does not have to be propagated through the tree as it is sufficient for the root node to check if there are any error messages in the collection attribute. In fact, in the new implementation receiving the compilation context and checking for error messages among the compilation messages is all that is done.

The Interpreter modules received less attention and were conveniently kept as imperative code.

6 Evaluation

6.1 Proving Extensibility

The *sup* notation is a construct that makes it possible for a sub-workspace to access its enclosing environment. It is similar to Java's *super* that is used to access overridden functions. *sup* was available in the G2 toolboxes and was always required when accessing the enclosing environment. JGrafchart on the other hand uses scoping rules similar to those of ordinary programming languages, making the *sup* notation unnecessary in most cases. It is however still useful for example when there is a local variable with the same name as a variable in the enclosing environment, see Figure 3.

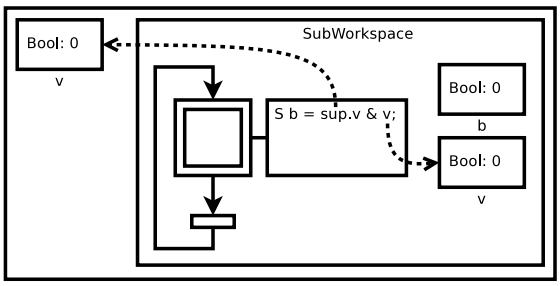

Fig. 3. Variable binding with *sup*.

After rewriting the language implementations, adding the *sup* notation as an extension was straightforward and only required 53 lines of code, of which 20 lines for the scanners and parsers, and 33 lines for the semantics.

Note that the *sup* extension only affects the lookup and thus only the Compiler modules are affected. Other extensions might also extend or alter the interpreter behavior. Extending the Interpreter would be trivial and altering it would also be possible by using JastAdd's *refine*.

6.2 Code Size Comparison

A metric that is commonly used to compare the size of software programs is source lines of code (SLOC). The metric has many disadvantages and must be used with care. It has been chosen just to show the difference in implementation size between the old and the new implementation. For simplicity all lines were counted regardless of whether they were statements, comments, or empty lines.

When writing the new implementation the focus has been to make it as understandable and maintainable as possible. An equivalent new implementation was created by making it equally compact as the old implementation, i.e. with a similar part of comments and empty lines. This is denoted New_{eq} in table 1.

Since the functionalities in the old implementation are mixed up, both with each other and with the generated code, distinguishing what is what was far from trivial. To determine which lines corresponded to which functionality a thorough analysis of the old implementation was performed.

To make the comparison as fair as possible, files and functions that were found to be dead code are not counted to their otherwise corresponding functionality and files that only contain generated code is considered a separate category. Import statements in the old implementation have been counted separately while the includes in the new implementation have been counted with the corresponding module.

Table 1. SLOC comparison of the old and the new implementation. New_{eq} is equivalent to New, but written equally compact as Old. In Total_{excl} Built-ins are considered as libraries, and are thus excluded. In Total_{fair} Separate Generated has also been excluded.

	Old	New	New_{eq}
Compiler	1537	380	209
Interpreter	1389	1089	983
Built-ins	3462	3514	-
Utilities	110	134	126
AST	2	114	52
Includes	230	-	-
Mixed Generated	1513	-	-
Separate Generated	6628	-	-
Dead	723	-	-
Total	15594	5231	-
Total_{excl}	12132	-	1370
Total_{fair}	5504	-	1370
sup	-	53	38

The AST specification is required by the JastAdd tool to be able to verify that the equations are valid. In the old language implementations the AST structure was mostly implicit according to the parser specification together with the JJTree stack implementation. The two lines that are counted as AST specification in the old implementation are modifications that had been made to the generated code to specify a non-implicit AST structure.

The new Compiler modules are 75% smaller than the old ones. Also the new implementation is not as compact as the old one. Comparing the old implementation with the equally compact New_{eq} implementation is even 86% smaller. Another thing to point out is that the new Compiler modules have been enhanced with several new compiler checks and additional attribution has been added to make the interpretation easier.

The Interpreter modules have not received much attention. The main reason why they are now smaller is that duplicated code has been removed. With all

the interpreter code gathered the code duplication was easy to detect and eliminate. The Interpreter modules in the new implementation is also somewhat less compact than the old implementation and the new implementation has also been enhanced, e.g. multiple dereferences within an expression is now supported.

In the old implementation the built-in functions and methods were implemented in both the actions and the conditions language. In the new implementation there is only one implementation but it has been split up into public classes for better maintainability. Earlier they were anonymous classes which require considerably less overhead. This is the reason why the new implementation is much larger than, as would be expected, half the size of the old implementation.

The mixed generated code lines in the old implementation make up roughly 20% of the lines in the manually maintained mixed files. With the new implementation no generated files have to be maintained.

The most fair comparison should be Total_{fair}, where the implementations are equally compact, and separate generated files and built-ins are excluded. Then the new implementation is then 75% smaller than the old one.

6.3 Performance Comparison

The compilation code of the old and the new implementation of JGrafchart were instrumenting manually, compilation was performed 100 times in a burst, and the best compilation time of these was considered. The Online Tutorial application was used since it is fairly large and contains a wide variety of constructs.

The compilation time was 17.3 ms and 39.3 ms for the old and the new implementation respectively. The new implementation performs more checks and it has also been rewritten to use a more extensible and maintainable, but worse performing name lookup. The rewritten lookup alone added 7 ms. Still, the new implementation takes roughly twice as long as the old implementation.

Interpreted performance has also been analyzed since it is currently the only way to execute JGrafchart applications. Profiling of the interpreters was also performed on the Online Tutorial, with the scan cycle time reduced to 10 ms. The execution code was instrumented manually and the execution time was added during approximately 5.7 million scan cycles. The average execution time per scan cycle were 0.204 ms and 0.212 ms for the old and the new implementation respectively. The execution performance is practically the same with the new and the old implementation. Better handling of dots and references weigh up the performance loss due to larger overhead and the new lookup. Lookup is involved since dereferencing performs dynamic name lookup during execution.

7 Summary

To enable further research on HLV, it must be supported by JGrafchart, and it is desirable to also keep a pure BV. Since HLV is a superset of BV it is desirable to re-use the BV implementation as a base, meaning that all aspects of the BV implementation must be extensible. ReRAGs and JastAdd have been used to

make the JGrafchart implementation of the textual sub-languages extensible. To confirm extensibility, the *sup* notation was added as an extension to the rewritten implementation of BV. The *sup* extension was easy to add and required only a few lines of code.

This confirms that using ReRAGs is a good way to create extensible compilers. In summary the pros and cons of the new implementation are:

- Extensibility
- Improved maintainability
- Modularized functionalities
- Fewer lines of code
- Increased robustness
- Degraded compiler performance
- Developers must understand attribute grammars

Improved maintainability and increased robustness lead to fewer bugs and better quality which in most cases is more important than performance.

In conclusion, the new JGrafchart implementation of the textual sub-languages should be ready for implementation of HLV as an extension. Related future work include making an extensible compiler for the function chart sub-language, investigating how to extend the editor in a suitable way, adding HLV as an extension, and finally evaluating new constructs for HLV.

Acknowledgements. Financial support from the Swedish Research Council through the LCCC Linnaeus grant is gratefully acknowledged.

References

1. Åkesson, J.: Optimica—an extension of modelica supporting dynamic optimization. In: Proceedings of Modelica'2008. Bielefeld, Germany (March 2008)
2. Ekman, T.: Design and implementation of object-oriented extensions to the Control Module language. In: Proceedings of NWPER'2004. Turku, Finland (August 2004)
3. Ekman, T., Hedin, G.: The jastadd extensible java compiler. SIGPLAN Not. 42, 1–18 (October 2007)
4. `http://www.gensym.com/product/G2` as of 2012-09-05
5. Gerber, T., Theorin, A., Johnsson, C.: Towards a seamless integration between process modeling descriptions at business and production levels - work in progress. In: Proceedings of INCOM'2012. Bucharest, Romania (May 2012)
6. Hedin, G.: An introductory tutorial on jastadd attribute grammars. In: Generative and Transformational Techniques in Software Engineering III. Springer (2011)
7. Johnsson, C.: A Graphical Language for Batch Control. Ph.D. thesis, Department of Automatic Control, Lund University, Sweden (Mar 1999)
8. Knuth, D.E.: Semantics of context-free languages. Theory of Computing Systems 2(2), 127–145 (June 1968)
9. Olsson, R.: Batch Control and Diagnosis. Ph.D. thesis, Department of Automatic Control, Lund University, Sweden (June 2005)
10. Theorin, A., Ollinger, L., Johnsson, C.: Service-oriented process control with grafchart and the devices profile for web services. In: Proceedings of INCOM'2012. Bucharest, Romania (May 2012)