Institutionen för datavetenskap
Department of Computer and Information Science

Final Thesis

OpenModelica Support for Figaro Extensions Regarding Fault Analysis

by

Alexander Carlqvist

LIU-IDA/LITH-EX-G--14/042--SE

2014-06-10

Linköpings universitet
SE-581 83 Linköping, Sweden

Linköpings universitet
581 83 Linköping

Linköping University
Department of Computer and Information Science

Final Thesis

OpenModelica Support for Figaro Extensions Regarding Fault Analysis

by

Alexander Carlqvist

LIU-IDA/LITH-EX-G--14/042--SE

2014-06-10

Supervisor: Lena Buffoni
Examiner: Peter Fritzson

Abstract

The practical result of this thesis is an extension to OpenModelica that transforms Modelica into Figaro. Modelica is an equation-based object-oriented modeling language. OpenModelica is an open source implementation of Modelica. Figaro is a language used for reliability modeling. Figaro is a general representation formalism that can be transformed into reliability models like fault trees. Figaro was designed for fault analysis. Modelica was designed to model the behavior of physical systems and run dynamic simulations. Because of that, you cannot just break components and analyze what happens to a system. This work enables us to have fault analysis in OpenModelica by transforming our Modelica model into a Figaro model and invoke the Figaro compiler. This lets us break particular components and see what happens to the system. This work is part of an ongoing effort to integrate several modeling environments.

Table of Contents

1 Introduction

The goal of this thesis is to transform Modelica into Figaro. This will be done by a mixture of translating a program and extracting information from the program. This transformation is to be an extension to the OpenModelica compiler.

Modelica is the source language. Modelica is an equation-based object-oriented modeling language [1]. It is used for dynamic simulations. The source language is the language we continuously have to deal with. Chapter 4 talks about how you describe a programming language. When talking about a programming language, we most often implicitly assume Modelica. The reader should already know Modelica well. The extension is about going from something known to unknown. Thus, we will not delve into the details of Modelica.

Figaro is the target language. Figaro is a logic-based modeling language [2]. It is meant for fault analysis, which essentially is what we want to add to OpenModelica. Chapter 2 talks about fault analysis, Figaro, and the reason for the extension.

In order to understand what the transformation will be, we have to go through the basic theory until we reach the point where we have a representation of the program that is favorable to do work on.

The basic idea behind a compiler is to take a program in some programming language and transform it into another programming language. In the end, you often have transformed a program written in a high-level programming language into a program in a low-level programming language. The most likely reason is you want to make an executable program from the program in the low-level programming language. The compiler is divided into several phases. For obvious reasons, we will only talk about phases that run before the transformation does. The point is to motivate why the transformation takes place when it does.

The analysis phases are the relevant ones. They adhere to the definition of the programming language. If the analysis phases succeed, the program is in a sense good. They must succeed before any final transformation can be done. The reason is each analysis phase asks the question whether some condition is fulfilled or not. If not, there is no point in going on trying to transform the program because it violates the definition of the programming language. Each condition is a particular kind of problem. We give a brief introduction to automata and computability theory, because each condition is tied to a certain kind of formal language and automaton. Chapter 3 introduces basic concepts. The idea is to show why the different analysis phases of a compiler are divided the way they are. Chapters 5, 6, and 7 introduce the theory behind the analysis phases.

Lexical and syntax analysis are important for one more reason. They raise the representation of the program. If one succeeds, it leaves a new and more abstract

representation of the program. After syntax analysis, we have our program represented as an abstract syntax tree together with an environment. Relate this to the lambda calculus. We now have a representation that is strong enough to capture everything that is computable. Thus, we have the representation we will do the actual work on.

Semantic analysis is the last check we can do. There are other questions we would like to answer, but because of limitations in computability, we are unable to do so. If we were able to answer them, we would have another condition that would have to be fulfilled before any transformation. This would be the imagined behavior analysis in Chapter 8.

Once the analysis phases are successfully done, we have validated the program as much as possible, and we are left with a representation of it that we are able to evaluate. This is the point where the extension comes into play. It is mainly an evaluation of the abstract syntax tree. We define what certain constructs should be in Figaro. Other data are extracted from the program when we stumble upon them. After evaluation, we have Figaro code. See Chapter 9 for more details.

The last thing the extension does is to invoke the Figaro compiler. The Figaro code is more or less a list of Figaro objects. The Figaro compiler takes this code together with some database. The Figaro compiler assembles a full Figaro 0 model from the input. The Figaro compiler can also use this model to make a fault tree. Chapter 10 talks about compiling Figaro and how we check for Figaro errors.

Chapter 11 gives an intellectual summary of the steps we take when transforming some representation into another. The reason is we are repeatedly exposed to problems of this nature and the approach to tackle them is a recurring theme.

This report is written in such a way that the main parts are not too technical. The benefit is that the line of reasoning is more clear because theory is more solid. The technical details are gathered in Annex A. They are likely to become outdated. Important to know is that the extension was designed to be invoked by a button in the graphical user interface OpenModelica Connection Editor. Also, suggestions about future work can be found in the annex because they are related to technical matters.

Annex B puts theory into practice and gives another example.

Finally, a remark on the practical benefits of the extension. It results in a closer coupling of OpenModelica and Figaro tools. Say you are to do fault analysis using Figaro. Instead of making a Figaro model out of a Modelica model manually, the extension provides an integrated tool chain that both converts the model and invokes the analysis for you. Thus, the process becomes less error-prone. Also, you save time, and your models are easier to maintain.

2 Fault analysis and Figaro

Generally, failure analysis means analyzing data to answer the question what was the cause of a failure. Specifically, fault analysis is a kind of failure analysis that uses deductive reasoning. Formally, you will have a deductive system [3]. That is, a set of axioms (known facts accepted without proof) and a set of rules of inference. Deductive reasoning is the process of drawing conclusions from premises (axioms). It can be seen as a top-down approach because the process is generally iterated using drawn conclusions as premises. The axioms that you start with must of course be true, else deductive reasoning is meaningless because the point is to draw conclusions that are necessarily true. There are three relevant rules of inference: modus ponens, transitivity, and contrapositive. Modus ponens says that P implies Q, P is true, and therefore Q must be true as well. The transitivity rule (syllogism) says that P implies Q, Q implies R, and therefore P implies R. That is, you can take a direct step instead of going through a middle point. The contrapositive rule (modus tollens) says that P implies Q, Q is false, and therefore P must be false as well.

Fault analysis is mainly used to get a picture of the logical steps leading up to an undesired state in a critical system. Such a system can be a telecom network or nuclear power plant. The undesired states need to be known in advance. The idea is to learn how a system can fail in order to make the system more safe and reliable. Basically, you start with small events such as component failures and trace the deduction until it reaches an undesired state. This could be a nuclear meltdown. If such a state is reached, there is a need for a modification of the system. The intent of the modification is to make the state less probable. Observe the introduction of probability here. It might not be possible to get rid of the risk completely.

A reliability model represents the probability distribution of all possible combinations of component failures that lead to a system failure. Fault trees and Bayesian networks in general are examples of reliability models.

Instead of having several different reliability models, Figaro was developed by EDF (Électricité de France) to be a more general representation formalism. Figaro is a reliability modeling language. Although there is a strong connection to logic, Figaro is not entirely logic-based. It is influenced by object-oriented languages in the way you represent objects and their properties. Figaro is mainly used with its graphical user interface KB3. To do modeling in Figaro, there are essentially three steps. First, you develop a database (knowledge base). Second, you make a so-called list of objects that will be in the model. Third, you invoke the Figaro compiler to make a full model in Figaro 0 out of the list of objects with respect to the database. That is, you instantiate the objects. Figaro 0 is a sublanguage of Figaro. The full model can then be used to generate, say, a fault tree. There are several reliability models the Figaro compiler can make out of the full model. The extension to OpenModelica corresponds to making the second step and invoking the third step. The first step is not meant to be done often. A database should be developed in the Figaro graphical user interface. A

database describes general rules for handling a system topology of some sort. The Figaro compiler processes the list of objects according to the rules of the given database. This division is good for reusability and keeps things simple. Because of it, we will see we are fortunate enough not having to deal with Figaro at a very detailed level because we are aided by the Figaro compiler. This is beneficial, because it means the extension does not have to care about what correctness in Figaro is. Thus, the extension is more adaptable to change.

There is a main point why we want to use Modelica and Figaro together. Modelica was designed to model the behavior of physical systems and run dynamic simulations of such systems. This is based on differential equations. Figaro, on the other hand, was designed for static analysis of systems and evaluating the effect that a fault will have on a system. In other words, Figaro is meant for fault analysis. The idea is that we want to do fault analysis on a Modelica model. In Modelica, we can simulate faulty components by giving them bad behavior. Here follows a telecom network example to illustrate the problem. Too high resistance to a resistor associated with a link in a telecom network will result in the link breaking down. We can simulate the network and break a single link. The network will still be functioning due to backup circuits. If we go ahead and break a certain number of links, the network will be malfunctioning. The question is which links we need to break in order for the network to break down. In a complex Modelica model, this would be more or less impossible to see. If we were to break a link in a physical Modelica model of the network, the model itself would break. The solution is to make a Figaro model out of the Modelica model and use Figaro to do fault analysis and answer the question which links to break. In conclusion, the extension to OpenModelica is a means to temporarily analyze a system modeled in Modelica statically using Figaro. See Figure 4. The results can then be investigated in a dynamic simulation.

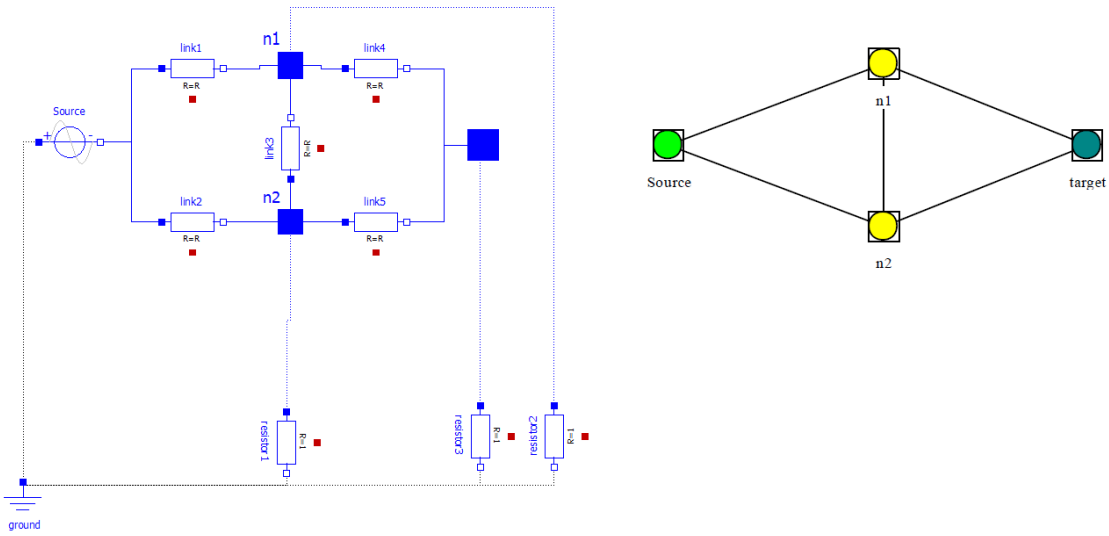

Figure 1. The telecom network example. To the left, we see the Modelica model. To the right, we see the Figaro model.

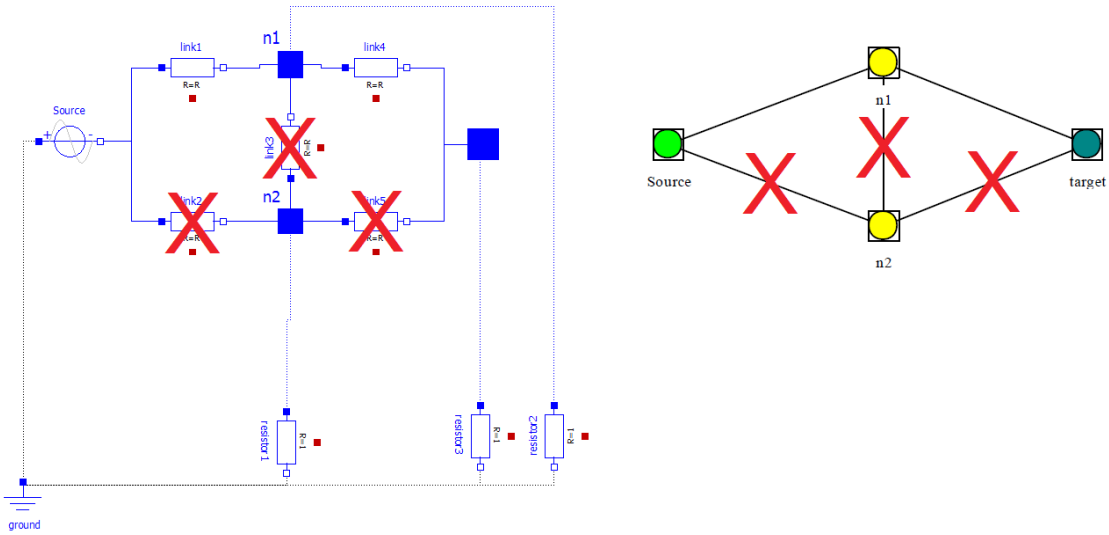

Figure 2. There are three component failures, but there is still a way from the target to the source. Thus, there is no system failure.

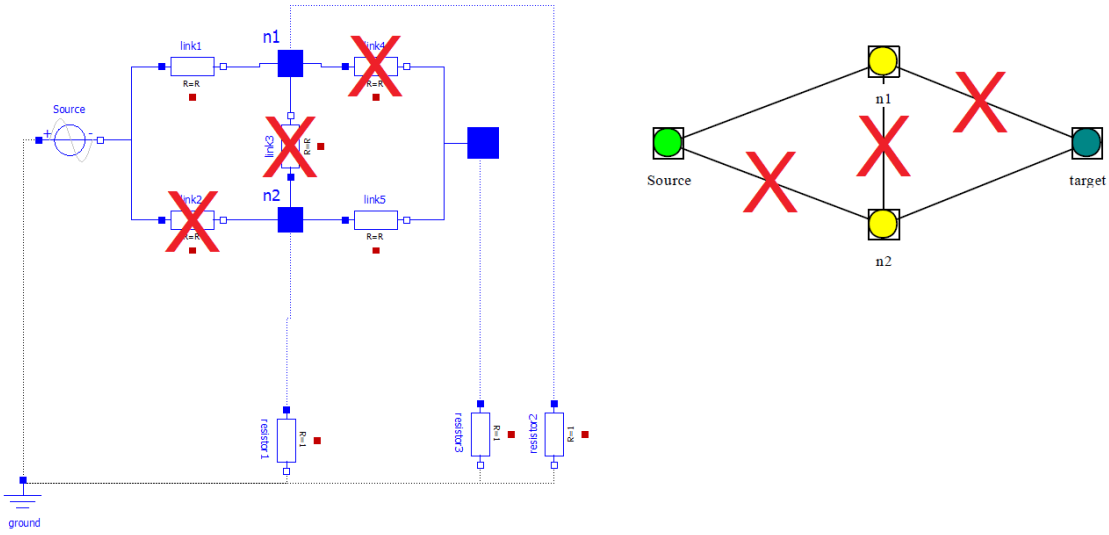

Figure 3. There are three component failures, and there is no way from the target to the source. Thus, the system is in an undesired state and we have a system failure.

An important remark is the different degrees of abstraction. In Modelica models, you must model all necessary physical components for natural reasons. Models meant for fault analysis are simplified. This has to do with the foundation in logic. If a component is deemed not to be important, it will be assumed to have ideal properties. Thus, there is no reason to represent it. You only keep facts that are relevant. This is the reason Figaro models made out of Modelica models will only contain some data from the Modelica models.

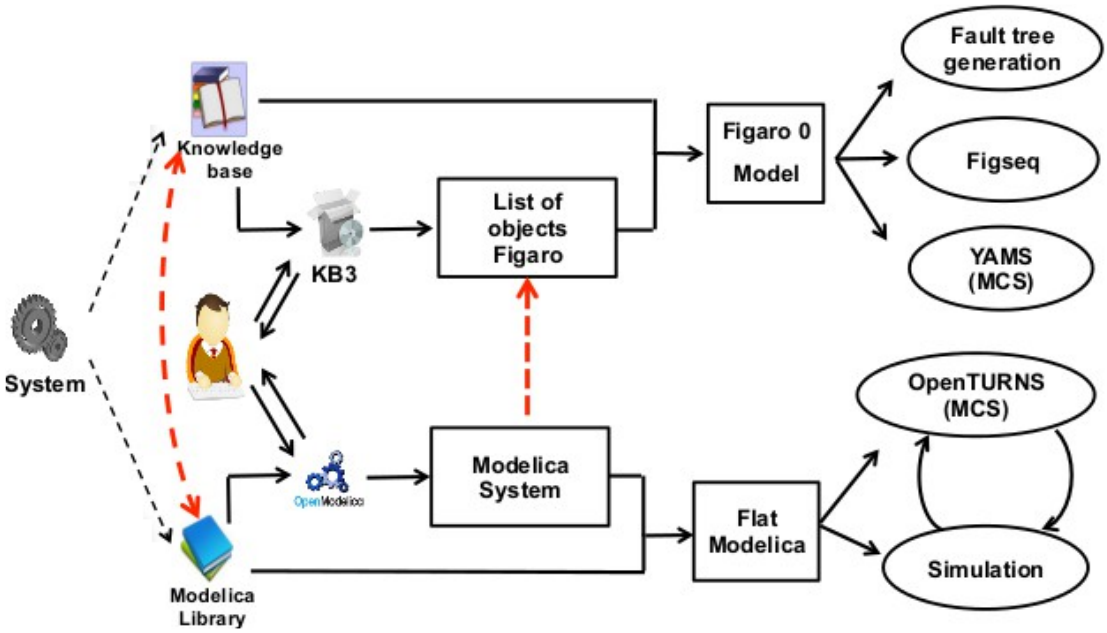

Figure 4. The big picture. The red link in the middle is basically the extension to OpenModelica.

In summary, with the extension to OpenModelica, we are connecting two different but complementary modeling techniques to enhance system reliability and ensure a coherent behavior of a system with respect to a physical model and a logic-based model.

3 Languages and automata

A few terms need to be stated. A symbol is an atomic unit. The alphabet Σ is a finite set of symbols. A string over Σ is a finite sequence of symbols from Σ. The language L over Σ is a set such that $L \subseteq \Sigma^*$, where Σ^* is the set of all strings over Σ.

We are concerned with problems. A problem often really is deciding whether some string is a member of some language [5]. If you want to be more precise, you can talk about decision problems [4]. However, we do not need such precision. Depending on the category of the language, the question of membership can be answered by different automata. Automata are mainly characterized by their ability to memorize and recall. Another characteristic is their control, but in general only deterministic control is of interest.

There is a hierarchy of languages we will deal with. The hierarchy resembles the Chomsky hierarchy [6]. See Figure 5.

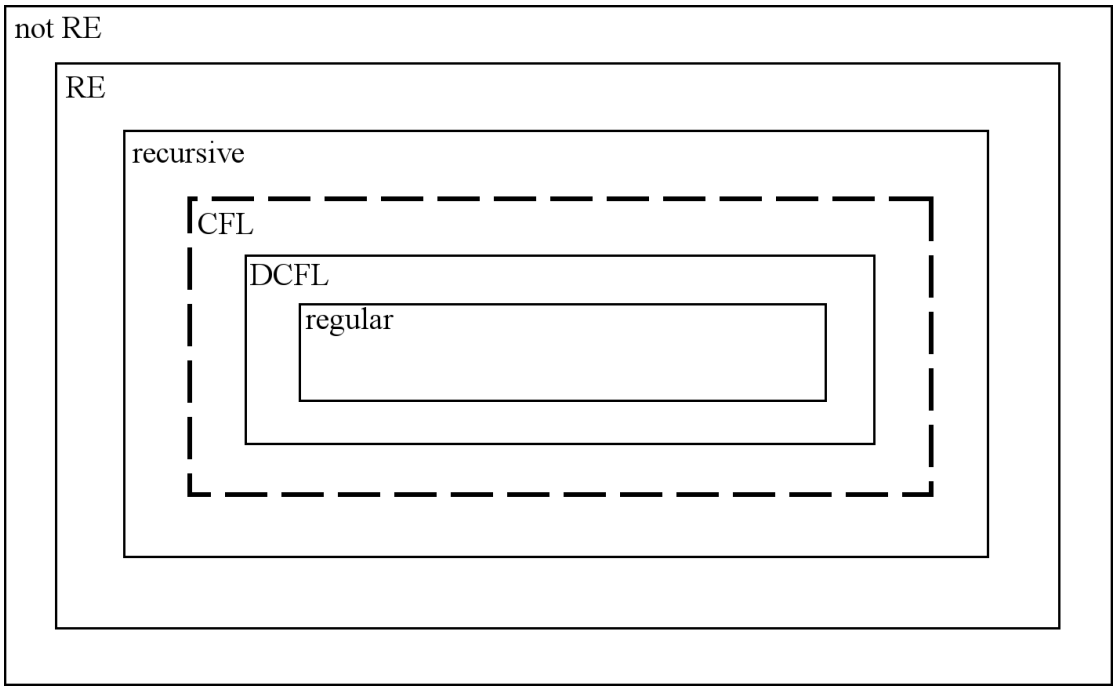

Figure 5. Sets of languages and how they are related. Solid lines for the sets of languages we are concerned with at some point.

Regular languages are the most simple class. These languages can be recognized by deterministic finite automata (DFA). A deterministic finite automaton is a machine with a finite set of states and it cannot be in more than one state at any time. Memory is therefore very limited. Nondeterministic finite automata (NFA) also recognize regular languages. A nondeterministic finite automaton may be in more than one state at any time. In the nondeterministic case, computation is not unique. Deterministic

and nondeterministic finite automata are equivalent in strength. That is, they can be transformed into one another. Regular languages are conveniently expressed using regular expressions [7]. A regular expression can be transformed into a finite automaton.

Deterministic context-free languages (DCFL) are recognized by deterministic pushdown automata (DPDA). Deterministic pushdown automata are similar to deterministic finite automata but they also have a stack [8]. Although this addition gives them infinite memory, access is restricted to LIFO (last in, first out). DPDA and LR(1) grammars are equivalent in strength. That is, they are capable of describing the same languages. The same set of languages is covered by both. Thus, a greater number of lookahead does not let LR grammars describe more languages. In contrast, LL grammars are affected by the number of lookahead. Thus, LL grammars describe proper subsets of the deterministic context-free languages. LL languages are of practical interest to us.

Although context-free languages (CFL) are an important set of languages, we are not concerned with it because it is a proper superset of the deterministic context-free languages. Put differently, nondeterministic pushdown automata (PDA) recognize a wider range of languages than deterministic pushdown automata do. That is, they are not equivalent in strength. Therefore, it might not be possible to transform a nondeterministic pushdown automaton into a deterministic pushdown automaton. Such nondeterminism simply will not do.

Recursive languages are recognized by total Turing machines. These languages are related to decidability. A property being decidable is equivalent to the set of those strings having that property being recursive. If the set is not recursive, the property is undecidable [5].

Recursively enumerable languages (RE) are recognized by Turing machines. Turing machines have infinite memory and access is unrestricted. This is depicted as a tape. Turing machines are a formalism for capturing what is computable. There are other formalisms for the same purpose. E. g., lambda calculus, which is tremendously important. The advantage of talking about Turing machines and automata in general is that they manipulate strings over an alphabet. This is intuitive and relates well to the source code of a program. Other formalisms have other building blocks. Lambda calculus, for instance, manipulates lambda terms instead. All formalisms for capturing computability are computationally equivalent. That is, all of them are able to do the same computation. Church's thesis states that the commonality is the notion of effective computability [4].

There are languages that are not recursively enumerable, but these cannot be recognized by any machine in practice [5].

4 Grammars

We desire a means by which we can precisely describe a programming language. However, there are two sides of a programming language we need to describe. First, we need to describe the syntax of the programming language. That is, the form of a program. Second, we need to describe the semantics of the programming language. That is, the meaning of a program. Here, we are only concerned with describing the syntax. Exactly what we need is a means to describe deterministic context-free languages. However, we only have context-free grammars (CFG), but there are principles and modifications that narrow the set of languages that can be described. Context-free grammars as we know them were introduced by Chomsky [6].

We define a grammar $G = (N, \Sigma, P, S)$. N is a finite set of nonterminals (syntactic variables). Σ is a finite set of terminals (symbols). $N \cup \Sigma = \varnothing$. P is a finite set of productions such that $P \subseteq N \times (N \cup \Sigma)^*$. S is the start symbol such that $S \in N$.

A production (A, α) is written as $A \rightarrow \alpha$.

A derivation in one step means substituting the right-hand side of a production for the left-hand side. That is, replacing a nonterminal by a string. A derivation is an arbitrary number of such substitutions. Doing a leftmost derivation means always replacing the leftmost nonterminal in each step. In a rightmost derivation, you pick the rightmost nonterminal instead.

Any string derivable from the start symbol is called a sentential form. There is a special case that is important. If a sentential form consists only of terminals, we say it is a sentence. That is, no more derivation can take place. The language of the grammar is the set of all sentences.

A parse tree records how a sentence derives from the start symbol.

A grammar is ambiguous if there are more than one parse tree for at least one sentence.

A more convenient notation for context-free grammars is Backus–Naur Form (BNF). There are several variations, but essentially they strive for ease of typing. Typically, a nonterminal is written as <nonterminal>, a terminal is written as 'terminal', and ::= is used instead of $\rightarrow$. Historically, BNF meant Backus Normal Form, but this is deceptive because it is not a normal form [9].

5 Lexical analysis

Some regular expression describes exactly what it means for a program in the programming language to be lexically well-formed. The language of that regular expression is the set of all lexically well-formed programs in the programming language. See Figure 6. Some deterministic finite automaton recognizes that very regular language. Let a character be a symbol. The alphabet is the set of allowed characters. If the deterministic finite automaton accepts the source code of a program as a string in that language, we have a lexically valid program.

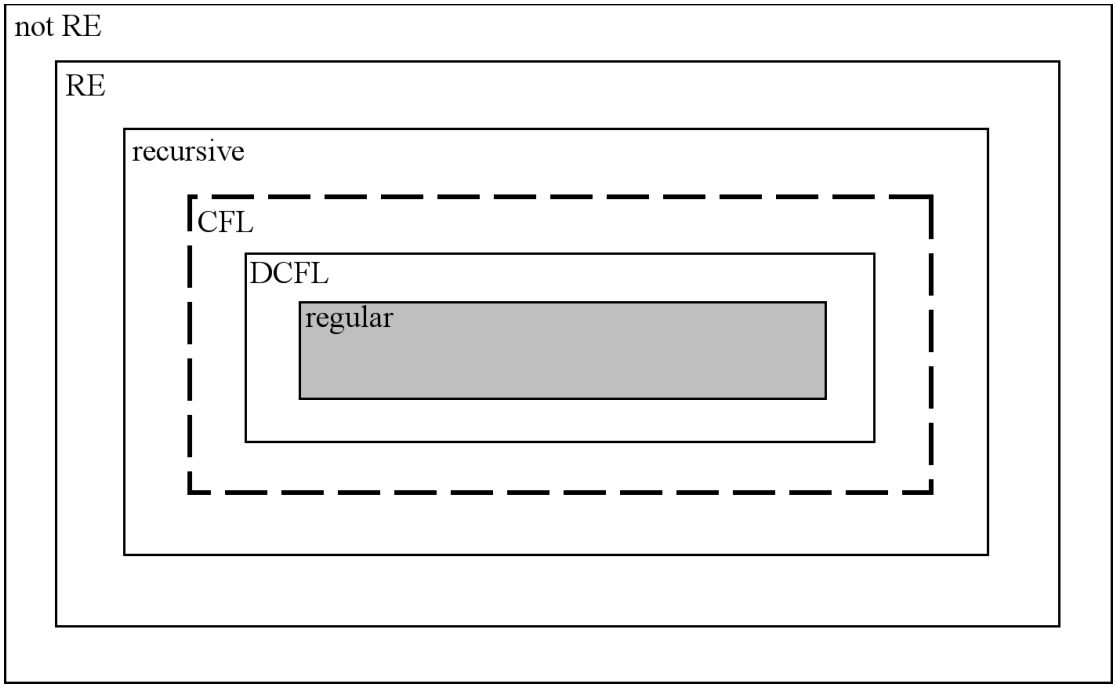

Figure 6. The regular language describing all lexically well-formed programs in the programming language lies somewhere in the shadowed area.

The source code of a program is no more than a sequence of characters. This format is rather unhelpful. Lexical analysis is about grouping characters that together have some meaning [10]. A sequence of such characters is called a lexeme. When the lexical analyzer identifies a lexeme, the analyzer makes a token. A token is a pair of a token name and a lexeme. The token name specifies the type of the lexeme.

Each type of lexeme is described by a regular expression. Regular expressions are closed under union. Therefore, the union of all those regular expressions is another regular expression. This regular expression describes the language in which the source code of a program is supposed to be a string. Regular expressions, nondeterministic finite automata, and deterministic finite automata are equivalent in strength. Therefore, the regular expression can be reduced to a deterministic finite automaton. Now we see the resemblance between a lexical analyzer and a deterministic finite

automaton.

Tokens are related to terminals of a context-free grammar. A lexeme is either a terminal or a sequence of terminals. If the lexeme is a single terminal, the token name will provide all information needed, and the lexeme is rendered unnecessary. If, on the other hand, the lexeme is a sequence of terminals, the token name will tell what nonterminal derived the sequence of terminals.

If lexical analysis is successful, we have a new representation of the program as a sequence of tokens.

6 Syntax analysis

There exists some deterministic context-free grammar (DCFG) that describes what it means for a program in the programming language to be syntactically well-formed. The language of that grammar is the set of all syntactically well-formed programs in the programming language. See Figure 7. There is a deterministic pushdown automaton that recognizes the deterministic context-free language defined by that grammar. Let a token name be a symbol. The alphabet is the set of all token names. If the deterministic pushdown automaton recognizes the program represented as a sequence of token names as a string in that language, we have a syntactically valid program.

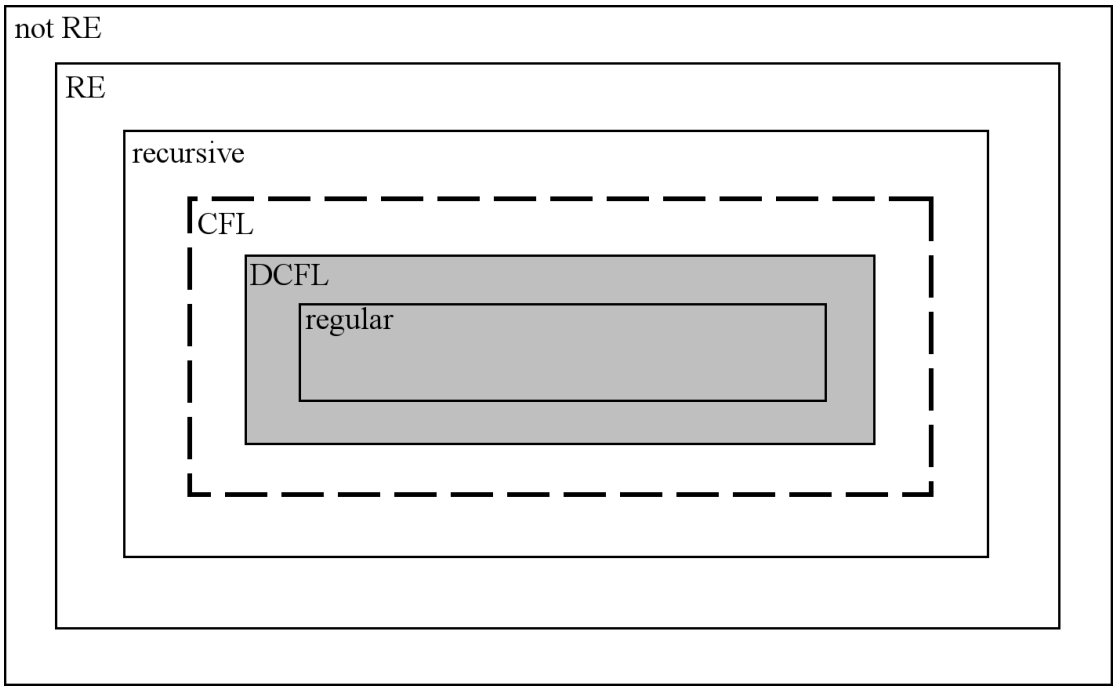

Figure 7. The deterministic context-free language describing all syntactically well-formed programs in the programming language lies somewhere in the shadowed area.

The objective of parsing is figuring out how to derive a given string of terminals from the start symbol of the grammar [10]. What that really means is figuring out all derivations. In principle, the parser constructs a parse tree (concrete syntax tree) giving all details.

There are two common strategies in parsing: top-down and bottom-up parsing [10]. In top-down parsing, you start constructing a parse tree at the root. You work your way down to the leaves. What this means is that you pick productions until you can derive the string of terminals from the start symbol. This is related to leftmost derivations. In bottom-up parsing, you go the other way around. That is, you start constructing a parse tree at the leaves. You work your way up to the root. Another way to put it is

that you look at what terminals you have and pick productions that can derive them until you finally pick one for the start symbol. This is related to rightmost derivations. We are only concerned with the former strategy because it is intuitive and a parser of that kind can be handcrafted. Parsers taking the latter approach are often generated. In fact, it would be unwise not to. It should also be pointed out that top-down parsers are not able to recognize all deterministic context-free languages. Some bottom-up parsers are.

A recursive-descent parser takes the top-down approach. There is a procedure for each nonterminal. Mainly, what we want to do is determining what production to apply for a nonterminal. The procedure essentially does three things. First, it picks a production. Then, for each nonterminal in the body, the procedure calls the procedure for that nonterminal, and for each terminal in the body, the procedure matches it with the corresponding input symbol. Should this matching fail, backtracking is required to try another production. A recursive-descent parser has the nice property that its stack is implicitly defined and managed by the call stack.

An LL parser is a top-down parser that reads the input from left to right and constructs a leftmost derivation. An LL(k) parser recognizes an LL(k) language, where k is the number of lookahead. An LL(*) parser does not use a fixed number of lookahead. Instead, such a parser makes a decision based on whether the following input symbols are accepted by a DFA or not. OpenModelica uses an LL(*) parser for Modelica generated by ANTLR.

If we take the components of a recursive-descent parser and restrict ourselves to some LL(k) grammar, we get a predictive parser. This is of particular interest for k = 1. Predictive means the parser needs no backtracking. That is, the parser always chooses the right production for a nonterminal based on the lookahead. Thus, only one pass over the input is required and input symbols may be thrown away once matched.

In practice, the parser is fed with a sequence of tokens instead of a string of terminals. It is undesirable to construct a parse tree, because it will be littered with symbols that no longer provide any meaning once in a tree structure. The parser may instead directly construct an abstract syntax tree (AST). The advantage is that nodes better reflect operations and operands. Also, the parser constructs an environment containing all bindings. Relate the abstract syntax tree together with the environment to lambda calculus.

Grammars easily become complicated and are typically written in a relaxed style that may allow unwanted strings. A grammar could be ambiguous. It could lack information about associativity or precedence of operators. The parser is in a good position to make decisions and could cope with these problems. Ambiguity could be resolved by disambiguating rules. Expressions could be handled by the shunting-yard algorithm [11].

7 Semantic analysis

There exists some recursive language that is the set of all semantically valid programs in the programming language. See Figure 8. Because the language is recursive, the property of a program in the programming language being semantically valid is decidable. Therefore, there is some total Turing machine that takes a string describing a program in the programming language. If the string is in the recursive language, the total Turing machine halts and accepts, which means we have a semantically valid program. Otherwise, if this is not the case, the total Turing machine halts and rejects. The point is that the total Turing machine always halts. That is, we always get a yes or no answer to the question whether a program is semantically valid or not.

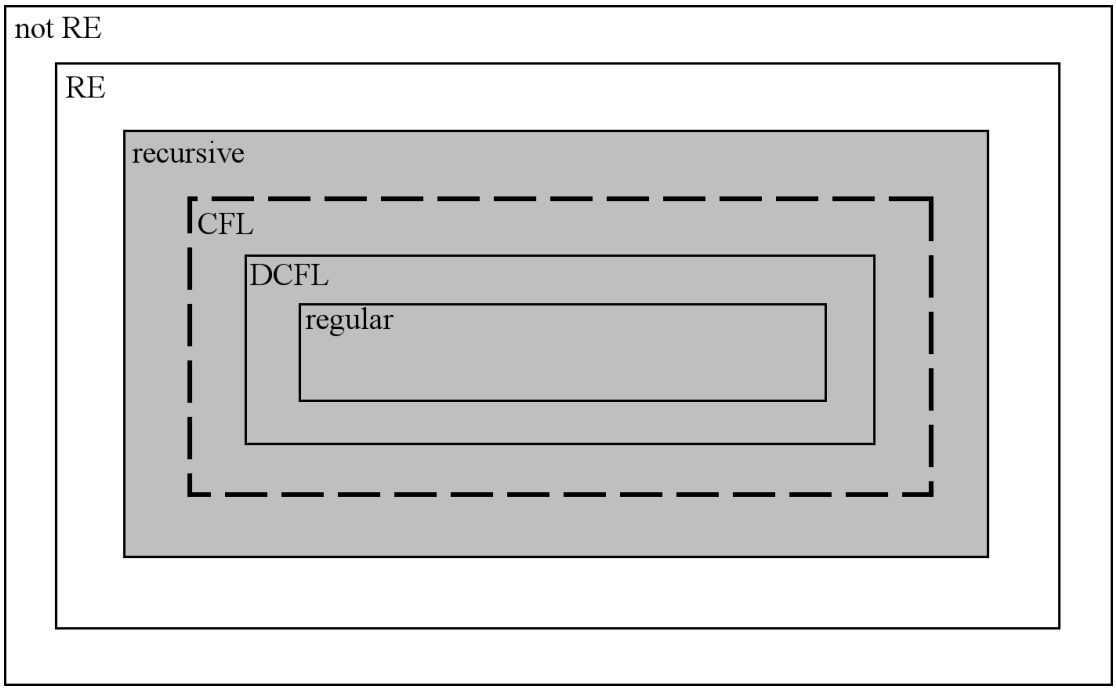

Figure 8. The recursive language describing all semantically valid programs in the programming language lies somewhere in the shadowed area.

Semantic analysis is of course not realized as a total Turing machine. It is hard to imagine what the symbols would be. The point is that a problem can only be solved by an algorithm if there is some total Turing machine for that problem [5]. Parsing outputs a representation of the program as an abstract syntax tree together with an environment. At earlier stages, we have seen different representations of the program from the perspective of formal languages and automata. From here on, we will in practice deal with tree problems.

Semantic analysis is about analyzing the meaning of a program to make sure operations do what they are meant to. For example, issues that need to be addressed are type checking and deciding whether variables are initialized before used.

8 Behavior analysis

There is some language of all programs in the programming language having the desired behavior. See Figure 9. If the language is recursively enumerable, the property of a program in the programming language having the desired behavior is semidecidable [4]. There is some Turing machine that takes a string describing a program in the programming language. If the string is in the language, the Turing machine halts and accepts. Thus, the program has the desired behavior. If, on the other hand, the string is not in the language, there are two possible outcomes. Either the Turing machine halts and rejects or it loops forever. Semidecidability is only of value from a mathematical point of view. If you were to run such a Turing machine and it gets stuck looping, you do not know whether it will eventually come up with an answer or not. That is, you do not really get an answer. This effectively makes the property undecidable [5]. If the language is not recursively enumerable, we are in the domain of languages that can only be recognized by oracle Turing machines [4]. That is, you need magic. The property of a program in the programming language having the desired behavior is therefore undecidable. We just motivate why the property should be undecidable. A typical question about behavior we might want to answer in order to catch undesired infinite loops is whether the program will eventually halt. This is the halting problem [12]. You can also show that the membership problem reduces to the halting problem [4]. As we have seen, problems are related to membership.

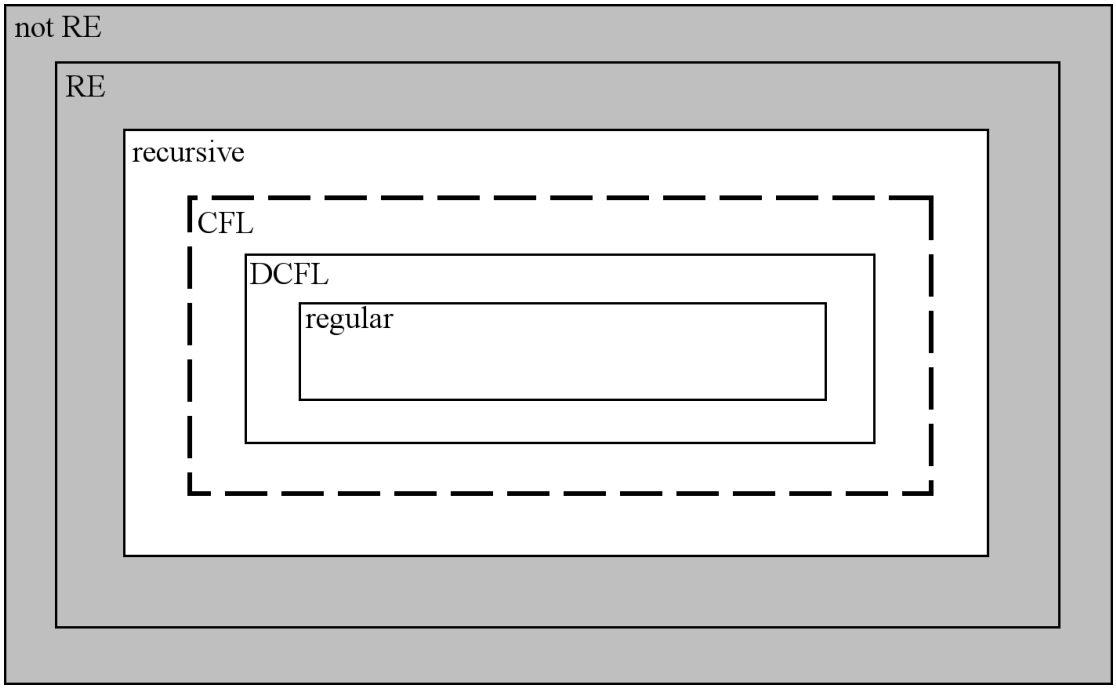

Figure 9. The nonrecursive language describing all well-behaved programs in the programming language lies somewhere in the shadowed area.

This illustrates that we cannot have a general analysis phase deciding whether a program has the desired behavior or not.

9 Evaluation

Here we treat a program as an abstract syntax tree together with an environment. To transform the program, we do an evaluation of the abstract syntax tree where the environment provides the bindings. We recursively define what the value of a construct should be. Most constructs have the empty value. Some attributes in the abstract syntax tree provide some values. The transformation of the program is the value of the program. That is really all to it. This is the principle of metalinguistic abstraction [13].

To evaluate, you do a depth-first traversal of the abstract syntax tree. That is, you start at the root and recursively visit unvisited children. The order is determined by the abstract syntax tree. Paraphrased, you go down as deep as possible as soon as possible. This suits our purpose well, because information belonging to a construct is generally represented as children of the node for that construct. Thus, we fetch the pieces we need, and then go ahead with the operation related to the node.

Functional programming is the natural paradigm for any work on a tree structure. In the beginning, there was Lisp. The paradigm as such was not important. Recursion was the important concept [14], and functional programming was done implicitly by recursion. Functional programming as a term and paradigm dates back to Backus's 1977 Turing Award lecture [15]. Functional programming is declarative. The beauty is to declare what to do and not how to do it. Simply put, when you need to solve a problem, you define a function for it. The computer is to evaluate that function as an expression in the common mathematical sense [16].

The extension for transforming Modelica into Figaro is written in MetaModelica, which is a Modelica-based functional programming language used to develop OpenModelica.

Here comes a description of the scheme we deploy to evaluate a program. For each kind of node that is to be visited in the abstract syntax tree, we define a function for visiting such a node. This function first calls the appropriate function for each of its children to get a value. That is, an argument. Then, the function applies the associated operation on those arguments to get the value of the node.

Some arguments to the visiting functions keep track of what state we are in. It is necessary to pass along such information because of static binding. If we had dynamic binding, there would be no need to carry this information as bindings would be determined by the call chain. Actually, the call chain reflects the path we are on in the tree. This is a motivation for why functional programming suits tree problems so well. Some state is implicitly handled by the call chain.

We want to find certain objects in a program. These objects are characterized by their types. There are two known abstract classes from which arbitrary classes may inherit.

The abstract classes have two attributes: a Figaro type name, and a piece of Figaro code. These attributes are just strings. The first attribute is supposed to be inherited from the base class if a class definition does not explicitly modify it. This could be seen as some sort of inheritance. The second attribute, however, should never be inherited as it merely is a piece of code which is to immediately follow the code generated for some object.

The first transformation is from the program into a sequence of ordered pairs of class names and Figaro type names. We traverse the abstract syntax tree to find class definitions that extend other classes, and classes introduced by short-hand notation. In the outmost call, the class should extend one of the abstract classes. If such a class is encountered, a recursive call is made to find those classes that extend that class, and so on. This is the point where we pass along the new Figaro type name if the encountered class modifies it.

The second transformation is from the program into a sequence of 3-tuples of component names, Figaro type names, and snippets of Figaro code.

The reason the first and the second transformation are separate is efficiency. There are only a few Figaro classes. Once they are extracted, the second transformation becomes more efficient because the abstract syntax tree needs only be traversed once. When a component declaration is encountered, a linear search is done to see if its type matches any of the class names in the result of the first transformation. If the first and the second transformation were merged, many more traversals would be required. Such an approach would lead to another major drawback; a loss in simplicity.

The third transformation is from the result of the second transformation into Figaro code. The value of the sequence of 3-tuples is a string of Figaro code.

10 Figaro compilation

The Figaro compiler takes Figaro code and a Figaro database. The Figaro code that we obtained from our transformation is more or less a list of Figaro objects. The Figaro compiler makes a full Figaro 0 model out of the code with respect to the given database. The Figaro compiler can run in different modes of operation. E. g., you can specify that the Figaro compiler should also convert the full model into a reliability model like a fault tree.

The Figaro compiler will give some information about the compilation in Extensible Markup Language (XML). XML is a language for structuring information. We want to extract error messages if there are any. That is, we want to transform the XML code into a sequence of error messages. Now we see another instance of the transformation problem.

We only need to recognize a subset of XML. We give a Backus–Naur Form (BNF) for the language we ultimately want to recognize. This grammar is an LL grammar. The start symbol is <answers>.

<answers> ::= '<ANSWERS>' <answer-list> '</ANSWERS>'
<answer-list> ::= <answer> <answer-list>
<answer-list> ::= ''
<answer> ::= '<ANSWER>' <error-list> '</ANSWER>'
<error-list> ::= <error> <error-list>
<error-list> ::= ''
<error> ::= '<ERROR>' <label> <criticity> '</ERROR>'
<label> ::= '<LABEL>' <characters> '</LABEL>'
<criticity> ::= '<CRITICITY>' <characters> '</CRITICITY>'

The lexical analyzer only needs to produce tokens for opening tags, closing tags, and text. The analyzer does recognize some characters that together have special meaning, but no tokens are created because they are simply not needed.

There is an intermediate phase in which unnecessary tokens are filtered out. We make a new sequence of tokens by walking over the one from the lexical analyzer. The purpose is to have a sequence of tokens that only contains tokens that are relevant for parsing. If we see an unknown opening tag, we walk past all tokens until we find its closing tag. Then we continue with the sequence of tokens after the closing tag. If we see a known opening tag but it does not directly contain important text, we walk past the first token of the rest of the sequence if the first token is text. If we see a closing tag, we do the same as in the previous case. When we have processed the sequence in this way, we only have key tokens the parser needs.

We do recursive-descent parsing on the final sequence of tokens. You could argue the parser is an LL(*) parser because of how match-expressions in MetaModelica work,

but in practice, it is a predictive parser. Parsing is actually mixed with evaluation because XML represents a tree. There is a function for each nonterminal that takes a sequence of tokens to parse. This function calls the appropriate function for each of its children to get a sequence of error messages and the sequence of tokens to continue with. Then, the function appends the sequences of error messages and returns the result together with the sequence of tokens to continue with. Functions use and walk past tokens as they see fit. Parsing starts by calling the function for the start symbol. This function only returns a sequence of error messages, which is the final value we seek.

11 Conclusions

There are both theoretical and practical conclusions. We begin with the theoretical ones, because they define the starting point of the practical work. When talking about a programming language, we implicitly assume our source language Modelica, which should be known to the reader. Figaro is the target language. As such, everything to do with Figaro is by definition. Therefore, when talking about value, Figaro code is meant.

We begin describing our programming language with a context-free grammar to get a feeling for what the language should be.

We look at our programming language from a regular language perspective. We describe what it means for a program in the programming language to be lexically valid with a regular expression. There is some deterministic finite automaton that recognizes the language of that very regular expression. This automaton lays the groundwork for the lexical analyzer.

We expand our view and look at our programming language from a deterministic context-free language perspective. We describe what it means for a program in the programming language to be syntactically valid with a deterministic context-free grammar. This grammar will largely resemble the context-free grammar for the programming language that we first made. However, they are not the same. There is some deterministic pushdown automaton that recognizes the language of the deterministic context-free grammar. This automaton is the basis for the parser.

We expand our view again and look at our programming language from a recursive language perspective. There is some recursive language that is the set of all semantically valid programs in the programming language. That is, it captures what it means for a program in the programming language to be semantically valid. There is also some total Turing machine that recognizes this recursive language. After all, semantic analysis is an algorithm that says yes or no to whether a program is semantically valid or not.

Generally, we would like to expand our view a final time and look at our programming language from a nonrecursive language perspective. The reason is to answer whether a program in the programming language has the desired behavior or not. However, a total Turing machine cannot recognize a nonrecursive language. In other words, we cannot have an algorithm that gives us a satisfactory answer.

Luckily, we do fine once we know the meaning of a program is valid. We recursively define what the value of a construct should be depending on what we want to do with the program. Then we just evaluate the program. The value of the program is the result we seek.

We have outlined the journey we take when dealing with a programming language up to a point. This is a recurring theme. Once we face new problems, they are often solved by adopting the same procedures. This is so because we work at the very foundation of computability.

The extension to OpenModelica works on the abstract syntax tree. That is, it runs when it is known that the meaning of a program is valid. Working on the abstract syntax tree offers a high degree of abstraction. This has been good during testing because you can easily track and see where an error occurs. Once the different cases and the abstract syntax tree were correctly interpreted, the extension turned out to be quite robust. In the beginning, it was hard to get a grip on what work was actually needed because of the loose specification, but once it was clear that such a few constructs needed to be dealt with, the task became easier. The first version of the extension worked on the Absyn (abstract syntax tree). That was kind of messy. The extension was rewritten to work on the SCode (simplified abstract syntax tree). This change reduced the size and made the code simpler. This was a good change to ensure the correctness of the extension. Because of the division of how modeling is done in Figaro, it became easier to verify the extension because you ended up with rather small files that needed to be identical. The extension works for the given examples and constructed dummy examples. It is important to keep in mind the specification for the extension was given by one example alone. This illustrates that the idea of what should be done was quite easy. Of course, understanding came gradually. The extension is more or less finished. There are some graphical tools to be added to make life easier for the end user, but they do not affect the core of the extension. Possibly, there could be new modes for the Figaro compiler in the future. In that case, these could be added to the extension.

Acknowledgments

I would like to thank everyone who has been involved in my work. Anders Haraldsson lent me some literature on functional programming. Hanna Sörensson gave me valuable feedback. Martin Sjölund gave me good technical advice. Adeel Asghar set up the buttons in the graphical environment. A few images related to Figaro are by courtesy of Marc Bouissou. My supervisor Lena Buffoni helped me with various issues. My examiner Peter Fritzson gave me good advice on writing.

References

[1] P. Fritzson, *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. NJ: Wiley-IEEE Press, 2004.
[2] M. Bouissou *et al.*, "Knowledge modelling and reliability processing: presentation of the FIGARO language and associated tools," in *Safecomp*, Trondheim, Norway, Nov. 1991.
[3] M. Ben-Ari, *Mathematical Logic for Computer Science*, 2nd ed. London: Springer-Verlag, 2001.
[4] D. C. Kozen, *Automata and Computability*. New York: Springer-Verlag, 1997.
[5] J. E. Hopcroft *et al., Introduction to Automata Theory, Languages, and Computation*, 3rd ed. Boston: Addison-Wesley, 2006.
[6] N. Chomsky, "Three models for the description of language," IRE Transactions on Information Theory, Sept. 1956, vol. 2, no. 3, pp. 113-124.
[7] S. C. Kleene, "Representation of Events in Nerve Nets and Finite Automata," in *Automata Studies*, C. E. Shannon and J. McCarthy, Eds. Princeton, NJ: Princeton University Press, 1956.
[8] T. H. Cormen *et al.*, *Introduction to Algorithms*, 3rd ed. Cambridge, MA: MIT Press, 2009.
[9] D. E. Knuth, "Backus Normal Form vs. Backus Naur Form," Communications of the ACM, Dec. 1964, vol. 7, no.12, pp. 735-736.
[10] A. V. Aho *et al.*, *Compilers: Principles, Techniques, & Tools*, 2nd ed. Boston: Addison-Wesley, 2006.
[11] E. W. Dijkstra, "Algol 60 translation : An algol 60 translator for the x1 and making a translator for algol 60," Stichting Mathematisch Centrum, 1961, no. MR 34/61, pp. 1-31.
[12] A. M. Turing, "On Computable Numbers, with an Application to the Entscheidungs problem," Proceedings of the London Mathematical Society, 1937, vol. 42, pp. 230-265. Erratum: Ibid., 1938, vol. 43, pp. 544-546.
[13] H. Abelson and G. J. Sussman, *Structure and Interpretation of Computer Programs*, 2nd ed. Cambridge, MA: MIT Press, 1996.
[14] J. McCarthy, "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I," Communications of the ACM, Apr. 1960, vol. 3, no. 4, pp. 184-195.
[15] J. Backus, "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs," Communications of the ACM, Aug. 1978, vol. 21, no. 8, pp. 613-641.
[16] R. Bird and P. Wadler, *Introduction to Functional Programming*. London: Prentice Hall, 1988.

Annex A

A.1 Compiling with exportToFigaro

The extension exportToFigaro has been developed for the bootstrapped version of the OpenModelica compiler (OMC). The transition to bootstrapping has unfortunately been delayed. In the meantime, a little fix is required. First, you build from trunk in an ordinary fashion. You must, else the built-in exportToFigaro will not be declared. Then, you need to edit CevalScript.mo. You are to introduce two lines of code that have been commented out. They are related to the script called Figaro. Just search for exportToFigaro, and you will see the comments. Once uncommented, you need to bootstrap the compiler. Now, your exportToFigaro will work as intended. When the transition to bootstrapping finally takes place, the code should be permanently introduced.

A.2 Relevant files

FrontEnd/ModelicaBuiltin.mo: Added exportToFigaro declaration.
Script/CevalScript.mo: Added two exportToFigaro cases.
Script/Figaro.mo: New file.
boot/LoadCompilerSources.mos: Listed Figaro.mo.
Util/Error.mo: Added a specific error message called FIGARO_ERROR.

A.3 Signature of exportToFigaro

The API function exportToFigaro takes the following arguments: path (TypeName), database (string), mode (string), options (string), and processor (string).

Argument path is the name of the class the extension will do the actual work on.

Argument database is the name of an XML file that specifies the database.

Argument mode decides what the Figaro compiler should do. There are two modes: figaro0, and fault-tree.

Argument options is the name of an XML file that specifies the options for fault tree generation. Only relevant if mode equals fault-tree.

Argument processor is the name of the Figaro compiler.

When exportToFigaro is done, it returns a Boolean constant telling whether the work was successful or not.

A.4 How exportToFigaro works

This example works up to the point when the Figaro compiler is to be invoked because there is no real database. The idea is just to illustrate the transformation that exportToFigaro does before the Figaro compiler does its part.

We have two abstract classes.

```
model Figaro_Object
	parameter String fullClassName;
	parameter String codeInstanceFigaro;
end Figaro_Object;

connector Figaro_Object_connector
	parameter String fullClassName;
	parameter String codeInstanceFigaro;
end Figaro_Object_connector;
```

We have the following package loaded.

```
package P
	model A
		extends Figaro_Object(fullClassName = "FigaroA");
	end A;
	connector B
		extends Figaro_Object_connector(fullClassName = "FigaroB");
	end B;
	connector C
		extends B;
	end C;
	connector D
		extends B(fullClassName = "FigaroD");
	end D;
	model E
		A a(codeInstanceFigaro = "figaroCode;");
		B b;
		C c;
		D d;
	end E;
end P;
```

We run exportToFigaro.

```
exportToFigaro(path = P, ...);
```

The following is the transformation.

OBJECT a IS_A FigaroA;
figaroCode;
OBJECT b IS_A FigaroB;
OBJECT c IS_A FigaroB;
OBJECT d IS_A FigaroD;

A.5 Graphical user interface

This section can be considered a manual on how to graphically call exportToFigaro.

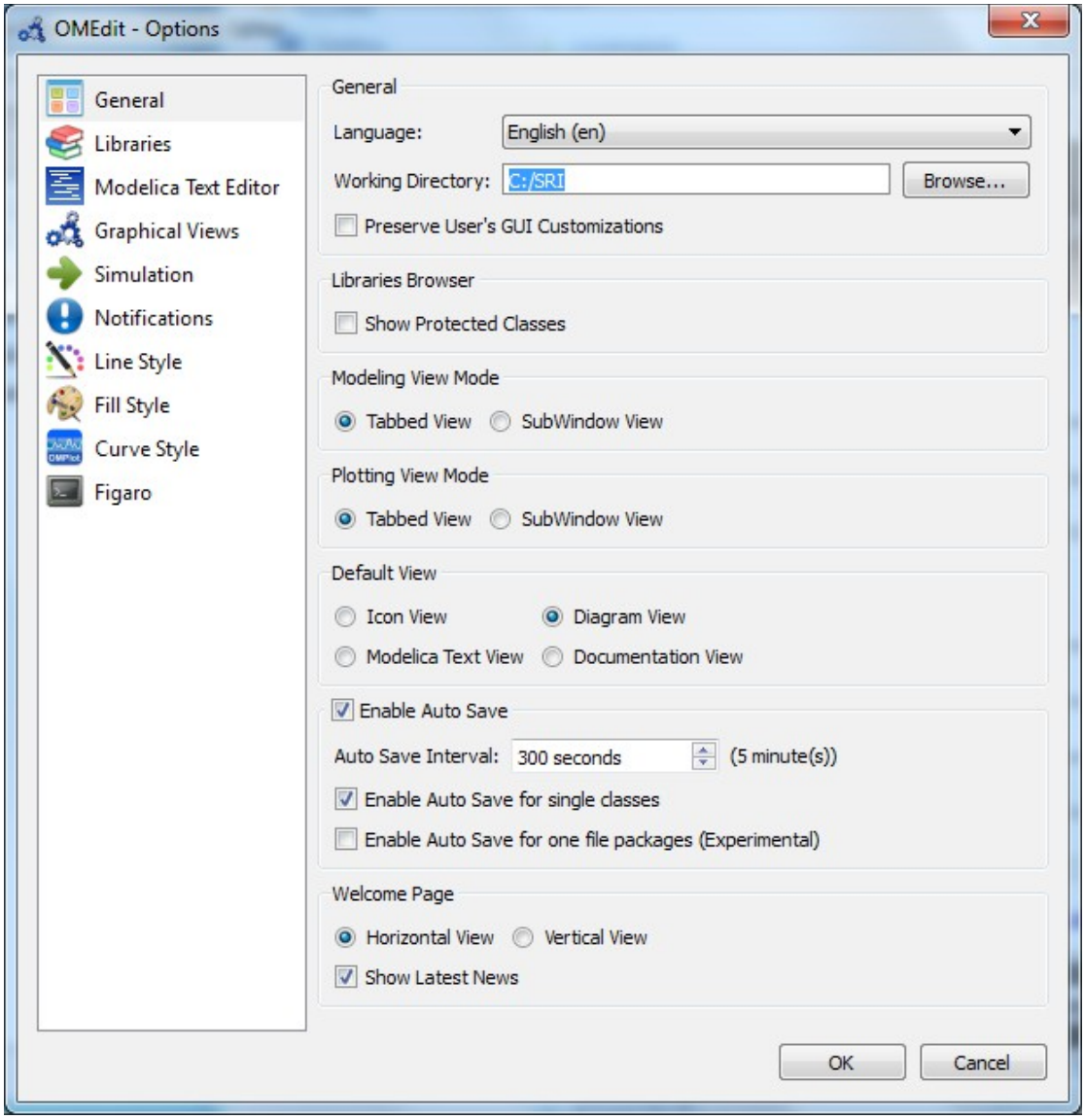

Figure 1. Set your working directory.

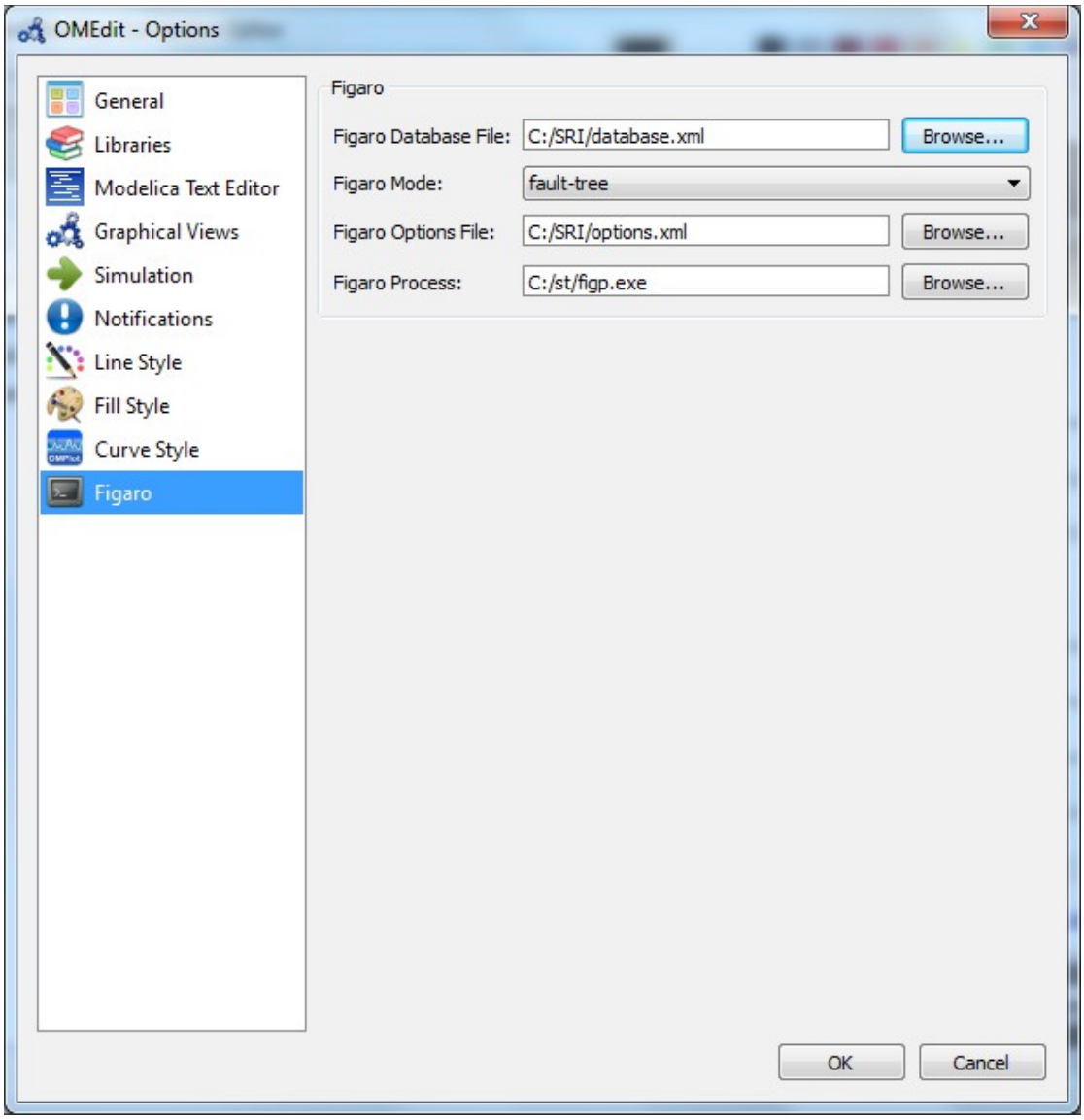

Figure 2. Choose database, mode, options, and compiler.

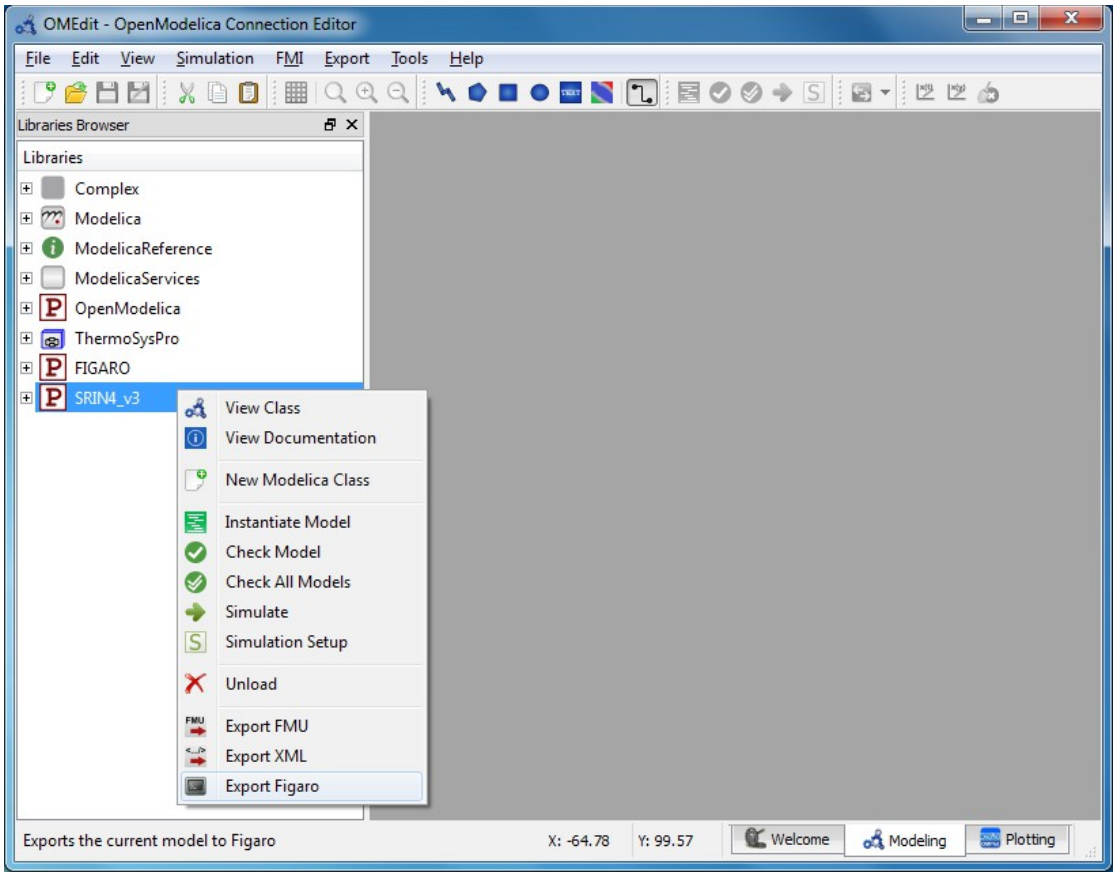

Figure 3. Right click on your package and click on Export Figaro.

A.6 Issues and decisions

Initially, the idea was to have a flag for OMC telling it to run the extension, but the extension fulfills the criteria of constant evaluation. Thus, an API function was a sensible choice.

The extension is supposed to be used in the graphical user interface OpenModelica Connection Editor. The extension should only work on the current active file in that environment. However, the environment will give a class name instead of a file name to the extension. This is because some loaded models might not be saved. The class name should in principle be the name of a package.

You must have the Figaro compiler installed before using the extension. It seemed undesirable to bundle the Figaro compiler with OMC. The Figaro compiler is invoked by a system call. Everything should work fine independent of your operating system.

The user should always set the working directory before using the extension. All files are written to and read from the working directory. The reason is the Figaro compiler will always put some temporary files in the working directory even though you tell it

to put the result files elsewhere. Using the working directory keeps all files in one place.

There are many side-effects because files need to be handled and the Figaro compiler needs to be invoked. Therefore, the code is organized in such a way that the different tasks and side-effects are all gathered in one top function. The benefit is that only this function is imperative and contains all dependencies.

The Figaro compiler works in an asynchronous way. A suggestion for the Figaro compiler is to add support for synchronous calls. That way the Figaro compiler will work better with OMC. The current fix is to sleep for three seconds after the Figaro compiler is invoked. If the sleep action is removed or the Figaro compiler would take longer than the sleep action, it is undefined what happens. The extension could either read the result file from a previous invocation of the Figaro compiler or the extension could fail because there is no file to read. That is, the success return value cannot be trusted.

The Figaro compiler puts some status information in an XML file. There are different kinds of errors that could be in that XML. Therefore, there was a need for a somewhat robust XML parser so you can easily change what messages to capture.

Annex B

SRI is a system for ensuring the cooling of parts such as pumps and alternators of nuclear power plants. As part of the thesis, an SRI model was slightly changed to provide an example of the use of the extension. Both a Modelica and Figaro model were given. The relevant Figaro database is called Skelbo. The work was a matter of figuring out what types the objects should have in Figaro and what interfaces were needed.

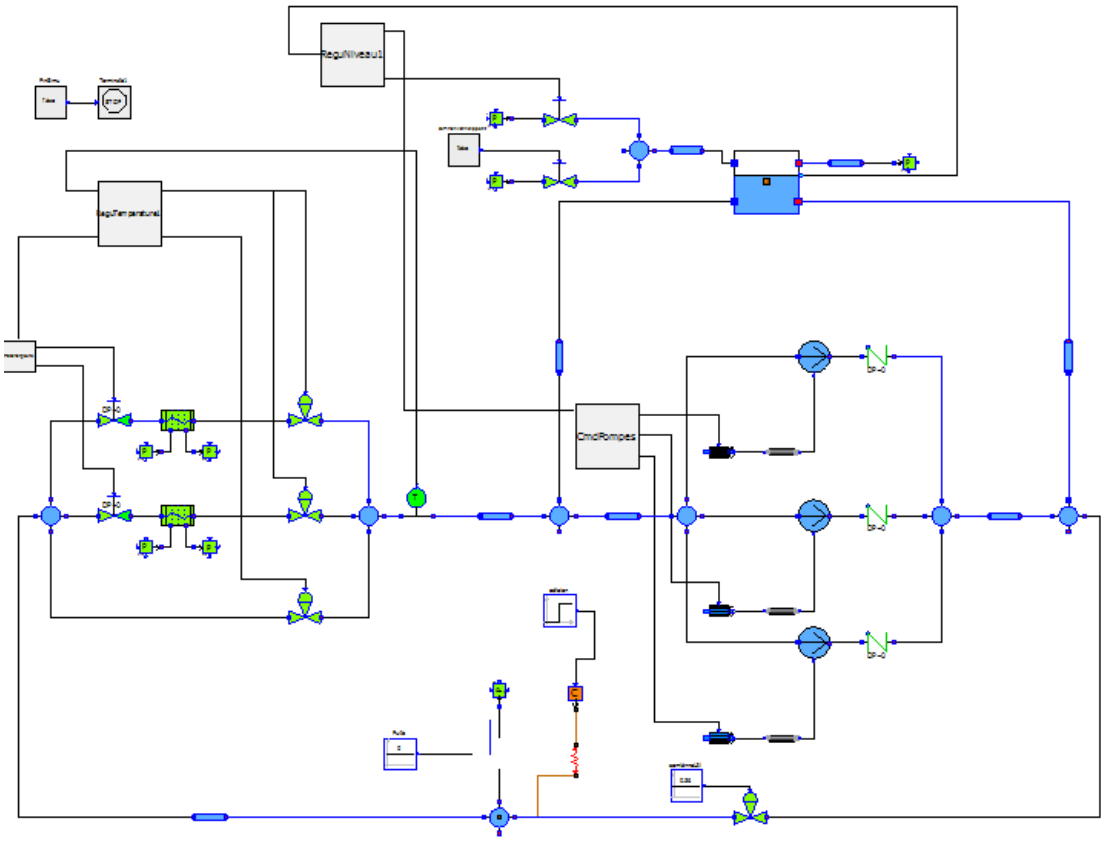

Figure 1. The Modelica model.

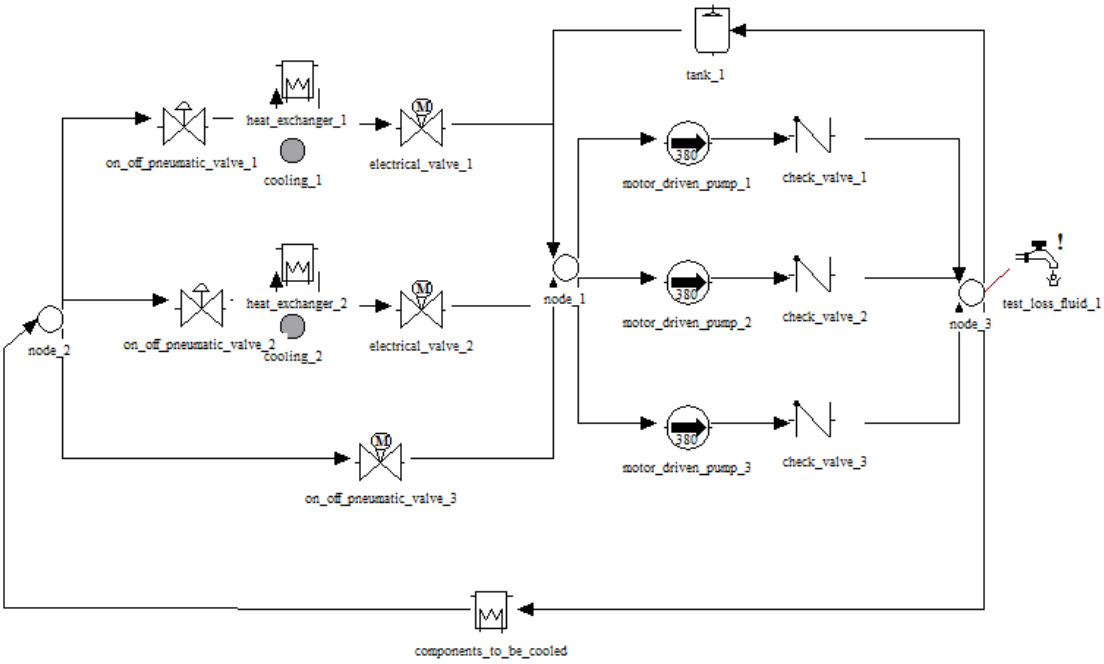

Figure 2. The Figaro model.

Note the similarities between the two models. The Figaro model contains less components because many are assumed to have ideal properties. We will now see an example of how the components in the Modelica model are to be modified. We study the tank. In the Modelica model, this is the big white and blue component in the top right area. In the Figaro model, the tank is labeled tank_1.

```
model Figaro_Tank
  extends FIGARO.Figaro_Object(fullClassName = "tank");
  extends ThermoSysPro.WaterSteam.Volumes.Tank;
end Figaro_Tank;
```

Figure 3. The class for tanks. We see there is a corresponding class in Figaro called tank.

```
  Figaro_Tank Bache1(A = 7, ze1 = 5, zs1 = 4.7, z0 = 3.11,
0)), Ce2(Q(start = -6.94)), codeInstanceFigaro = "INTERFACE
      upstream
      = VolumeA1;
      downstream
      = VolumeA2;") annotation(Placement(visible = true, tr
```

Figure 4. The instance. The interface says VolumeA1 is the source. This corresponds to the link from node_3 to tank_1 in the Figaro model.