The Design and Implementation of the μModelica Compiler

Weigao Xu
Supervisor: Prof. Hans Vangheluwe

School of Computer Science
McGill University, Montréal, Canada

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfilment of the requirements of the degree of
Master of Science in Computer Science

Abstract

Modelica is a recently developed object-oriented language for physical systems modeling. It is a modern language built on *non-causal* modeling with mathematical equations and object-oriented constructs.

An open source research prototype compiler for μModelica, a subset of Modelica, is presented. The compiler takes textual Modelica source as input, translates it into flat Modelica, then performs a series of symbolic transformations on the *differential-algebraic equations*, most notably, assigning causality, and generates input suitable for processing by a numerical simulator such as Octave.

Design and implementation issues of the μModelica compiler are discussed in some detail in this thesis. These issues include the general architecture of the compiler, semantic analysis, formula manipulation, and code generation. Some advanced formula manipulation techniques are also studied, and are proposed to be implemented as future work.

Modelica est un language orienté objet développé récemment dans le but de modéliser les systèmes physiques. C'est un language moderne, bâti à partir de la modélisation non causale, qui supporte des équations mathématiques et des constructions orientées objet.

Un prototype de compilateur libre de droit pour μModelica, un sous-ensemble de Modelica, est présenté. Un fichier Modelica textuel est envoyé au compilateur comme variable d'entrée. Celui-ci traduit le texte en Modelica simple et génre une série de transformations symboliques partir des *équations différentielles*. En particulier, la causalité est déterminée et du code pouvant être interprété par un simulateur numérique, tel Octave, est généré.

Les particularitées d'implémentation et de design du compilateur sont discutées en détail dans cette thèse. Elles incluent notamment l'architecture du dit compilateur, l'analyse sémantique, la manipulation des formules et la génération du code. Quelques techniques avancées dans la manipulation des formules sont aussi étudiées et l'implémentation de celles-ci est proposée comme avenue future.

Acknowledgments

I would like to express my sincere gratitude to all the people who have helped me and encouraged me during my study at McGill University, especially in the process of developing the μModelica compiler.

First of all, thanks to Prof. Hans Vangheluwe, my supervisor, who has activated my research interest in the field of Modeling and simulation, and has earnestly supervised my work in the development of the μModelica compiler.

Many thanks to my parents, for their love, and for supporting me in many aspects during my study.

Thanks to Peter Bunus at PELAB, Linköping University, who has kindly offered us a free Modelica parser. It has saved us a lot of time in the implementation of the Front End.

Thanks to Jean-Sébastien Bolduc, for his inspiring discussions during the weekly meetings for the μModelica compiler.

Thanks to Marc Provost for his research work in PyGK (a Python graph kernel), which enables the XML representation of Modelica models, and the transformation from C++ parse tree to Python parse tree.

Finally, thanks to the Quebec tax payers, for their contributions to the funding for this project (through the Fond de Recherche sur la Nature et les Technologies New Researchers fund)..

Contents

List of Figures

1 Introduction

1.1 Background

Modeling and simulation have been an important part of computing for a few decades. Computer simulation is used in industry to reduce the cost and time of development, and to optimize product design. As computer technology develops rapidly in recent years, the demand to simulate increasingly complex systems also grows.

In the past, *causal* models were most widely used n continuous, lumped parameter modeling of systems [8]. Causal models are commonly represented in the form of either *causal block diagrams* or in a *Continuous System Simulation Language* variant of the CSSL standard [24]. The semantics of such models is gieven by *ordinary differential equations* (ODEs). As systems under study become more and more complex, the requirement for reuse of components in modeling is getting increasingly important. Causal models are not very suitable for component re-use.

Over the last decades, numerous simulation tools have been developed. Some of these tools are general-purpose simulation tools, such as Simulink [23], which are based on causal (input/output) block diagrams. Other tools were developed for simulating models in specific domains, such as electronic components and mechanical devices. The major disadvantage of these tools is that they might be able to provide optimal methods in one domain, but are often not capable of representing and/or simulating structure and behaviour of systems in other domains. This precludes supporing multi-domain or multi-physics modeling.

To model and simulate increasingly complex and heterogeneous technical systems which consist of components from different domains, as well as to support meaningful model re-use, a new modeling language and supporting compiler were needed. In particular, the following problems needed to be solved:

- allow modelers to focus more on the description of the behaviour of system components, i.e. *non-causal modeling*, instead of spending a lot of effort on deriving a causal representations suitable for efficient numerical simulation;
- provide domain-neutral modelling and efficient simulation of multi-domain systems;
- support model reusability: the capability of creating easy-to-re-use components.

For the above reasons, in 1996, initiated by Hilding Elmqvist, a group of researchers from universities and industry started the development of a new object-oriented modeling language. The new language was called Modelica. It is a modern language built on non-causal modeling with mathematical equations and object-oriented constructs to facilitate reuse of modeling knowledge [5].

Comparing to other current modeling technologies, Modelica has the following advantages [16]:

- Object-oriented modeling. This makes it possible to create physically relevant and easy-to-re-use model components, which are employed to support hierarchical structuring, re-use, and evolution of large and complex models covering multiple technology domains.
- Non-causal modeling. Modeling is based on equations instead of on assignment statements as in traditional input/output block abstractions. Direct use of equations significantly increases reusability of model components, since components adapt to the data flow context in which they are used. This generalization enables both simpler models and more efficient simulation (thanks to global, symbolic, compile-time optimizations).
- Physical modeling of multiple domains. Model components can correspond to physical objects in the real world, in contrast to established techniques that require conversion to signal blocks. For application engineers, such physical components are particularly easy to combine into simulation models using a (possibly domain-specific) graphical editor.

1.2 An Overview of Modelica

As in the object-oriented programming language Java, the basic structuring element in Modelica is a *class*. Almost everything in the real technical world can be represented as a class, and the entire model is hierarchically composed in terms of classes. But the structure of a Modelica class is different from that of a Java class. A typical Modelica class has two parts, the declaration part, and the equation part. The declaration part contains declarations of variables, which are class attributes representing data. The equation part contains equations which specify the behavior, that is, the relationship between declared variables. Equations in Modelica are different from assignment statements in traditional languages. There is no causality assigned in Modelica equations which are "implicit". For example, equation $a = b + c$ can be written as $b + c = a$. The meaning of these two are equivalent. Also, equations can be written in any order.

1.2.1 Modeling an Electrical Circuit in Modelica

This section introduces the key features of Modelica through the example of an electrical circuit, which is shown in Figure 1.1.

This circuit consists of a set of inter-connected electrical components, which include a voltage source, two resistors, a capacitor, and a ground point. The following model is a Modelica description of the complete circuit:

```
model Circuit
  Resistor R1(r=1);
  Resistor R2(r=1);
  Capacitor C(c=1);
  VsourceAC AC;
  Ground G;
equation
  connect (AC.p, R1.p);
  connect (R1.n, R2.p);
  connect (R2.n, C.p);
  connect (C.n, AC.n);
  connect (AC.n, G.p);
end circuit
```

From this model, we can see that modeling in Modelica is very intuitive. System topology is con-

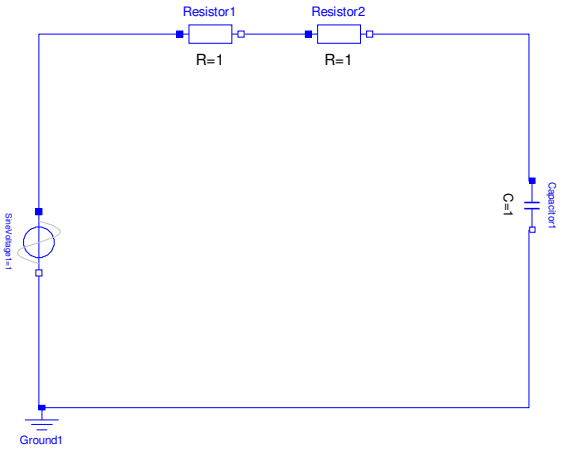

Figure 1.1: An Electrical Circuit

served by diving the whole system into components and linking these components by connections. The underlying meaning of a connection is again given in terms of equations. which specify the interaction between connected components.

The declarations of resistors, capacitor, etc. create instances of components. The definitions of these components are described in other classes.

1.2.2 Basic Language Elements

Modeling of a large system in Modelica is hierarchically broken up into a set of components, which should be reusable. Modelica has the following language elements to support this:

- Pre-defined types: Real, Integer, Boolean, and String. These are the basic components at the lowest level in Modelica;
- Structured components, enable hierarchical structuring
- component arrays, to handle matrices, arrays of submodels, etc.
- Equations and/or algorithms (assignment statements). Note that Modelica also supports causal modelling. In this thesis we will focus on the non-causal part of Modelica;
- Connections which couple model components.

1.2.3 Restricted Classes

Class is the fundamental structure element in Modelica. A class in Modelica can be defined using the keyword **class**. But under certain conditions, the keyword **class** can be replaced by one of seven

other, more specific keywords: **model**, **connector**, **record**, **block**, **type**, **function**, and **package**.

On the one hand, the restricted class mechanism makes Modelica code easier to read and maintain. It is also modeler-friendly since the modeler does not need to learn several different language constructs, but just the class concept. On the other hand, all properties of a general class are identical to all kinds of restricted classes. For example, the syntax and semantics of definition, instantiation, inheritance, and general properties are defined in the same way for all kinds of classes. Such *orthogonality* simplifies the construction of a Modelica compiler since only the syntax and semantics of the class construct, along with some validity checks on a restricted class need to be implemented.

The following summarizes the restrictions and usage of each kind of restricted class in terms of some examples.

model

The only restriction of a **model** restricted class is that it may not be used in connections. Its semantics are identical to the general class construct in Modelica, and it is most commonly used. The previous example `Circuit` is defined as a **model** class.

record

The **record** class is used to describe structured data. No equations are allowed in the definition or in any of its components. It may not be used in connections and may not contain protected elements. For example:

```
record Student
  String name;
  Integer studentNumber;
  String department;
end Student;
```

type

A type restricted class may only be an extension to the predefined types, enumerations, record classes, or array of type. Therefore, it can only be used in short class definitions to introduce new types. For example, the following type definition is illegal:

```
// Users can not define a new type
type Type1
  Real x;
end Type1;
```

The class definitions of `Voltage` and `Current` in section 1.2.4 show how new types are defined by means of short class definition.

connector

The restrictions of **connector** classes are identical to those of **record** classes, except that **connector** classes are designed to be used in connections. A connector example is given in section 1.2.5.

block

The **block** restricted class is used to model causal (input/output) block diagrams. In Modelica, the two keywords, **input** and **output**, are used as component prefixes to postulate the data flow

direction. All declared variables in a **block** must either have the prefix **input** or **output**. A **block** class may not be used in connections. A simple example:

```
block CircleAreaCalculator
  parameter Real pi = 3.14;
  input Real radius;
  output Real area;
equation
  area = pi * radius^2;
end CircleAreaCalculator;
```

package

Since Modelica supports nested class declarations, the **package** restricted class is designed to manage name spaces of classes. The restrictions of a package is that it may only contain class definitions and constant declarations, i.e., no variable or parameter declarations. Dot-notation is used to refer to inner classes. The following is a stripped-down example of package:

```
package Electronic
  constant Real pi = 3.1415926;

  connector Pin
    ...
  end Pin;

  model Resistor
    ...
  end Resistor;

  model Capacitor
    ...
  end Capacitor;

end Electronic;
```

function

The semantics of **function** classes is similar to that of **block** classes. In addition to the restrictions applied to the **block** classes, a **function** class is also restricted by the following rules:

- No equations and initial algorithms are allowed. At most one algorithm clause is allowed.
- Calling a function requires either an algorithm or an external function interface.
- No calls to the Modelica built-in operators **der, initial, terminal, sample, pre, edge, change, reinit, delay** and **cardinality** are allowed in a function as their arguments are time-varying signals as opposed to intantaneous values.

A simple example function:

```
function Add
```

```
  input Real x;
  input Real y;
  output Real result;
algorithm
  result := x + y;
end Add;
```

1.2.4 Types and Physical Quantities

Physical quantities are used to describe the properties of physical systems, e.g. *Voltage* and *Current* in the electrical circuit example. These quantities can be defined in Modelica in terms of restricted class **type**:

```
type Voltage = Real (unit="V");
type Current = Real (unit="A");
```

where *Real* is a pre-defined type. This is the short form of defining classes in Modelica. The above two definitions mean that Voltage and Current have the same definition as Real except that the attribute *unit* is modified.

In Modelica, pre-defined types, i.e. *Real, Integer, Boolean,* and *String* are not the primitives. The pre-defined types are classes built over primitives. For example, the conceptual definition of Real is given in [5]:

```
type Real
  RealType value;
  parameter StringType quantity = "";
  parameter StringType unit = ""     "unit used in equation";
  parameter StringType displayUnit = ""  "Default display unit";
  parameter RealType min = -Inf;
  parameter RealType max = +Inf;
  parameter RealType start=0    "initial value";
  parameter BooleanType fixed = true;  //default for para/const
                              = false;   //default for other vars
  parameter BooleanType enable = true;   //defined for every class
  parameter RealType nominal;
  parameter StateSelect stateSelect = StateSelect.default;
equation
  assert(value>=min and value<=max, "Variable value out of limit");
  assert(nominal>=min and nominal<=max, "Nominal value out of limit");
end Real;
```

where *RealType, IntegerType, StringType,* and *BooleanType* are the primitive types. But in order to avoid confusion, modelers start creating models from pre-defined types. The relationship between primitives and pre-defined types is handled internally by the compiler.

From the above class definition, we can see that Real has actually encapsulated a set of attributes, such as *value, unit, start,* etc., which make it well-suited for describing physical quantities.

1.2.5 Connections

In Modelica, models can be built up of components which are coupled by connections. Connectors are communication interfaces between components, over which they are connected to form coupled models. The connector class in the electrical circuit example is defined as follow:

```
connector Pin
  Voltage v;
  flow Current i;
end pin;
```

The meaning of a connection statement is given in terms of equations. A connection statement `connect(pin1, pin2)`, where `pin1` and `pin2` are instances of connector class **Pin**, connects the two pins such that they form a node. The meaning of this connection is equivalent to the following two equations:

```
pin1.v = pin2.v
pin1.i + pin2.i = 0
```

The physical meaning of the first equation is that, there is no voltage drop at a node. The second equation describes Kirchoff's current law.

In a connector class, a variable declared without the prefix **flow** is called an *across* variable, e.g. `Voltage v` here in this case. The conversion rule for connected across variables is that they are set equal. A variable declared with the prefix **flow** is called a *through* variable. Connected through variable are summed to zero at each node. Similar laws apply to flow rates in a piping network and to forces and torques in mechanical systems [4].

1.2.6 Partial Models and Inheritance

As in other object-oriented languages, there is a mechanism in Modelica to define an *interface* for different types of objects that have common properties. In the electrical domain, many components have two pins. An *interface* is defined as follow for these components:

```
partial model TwoPin "Superclass of elements with 2 electrical pins"
  Pin p, n;
  Voltage v;
  Current i;
equation
  v = p.v - n.v;
  0 = p.i + n.i;
  i = p.i;
end TwoPin;
```

In addition to the two pins, `p` and `n`, the model also includes two attributes, quantity `v`, which defines the voltage drop across the component, and another quantity `i`, that defines the current flowing through the component. This model introduces 4 variables and 3 equations. Therefore, it is an incomplete model. A constitutive equation must be added to make it complete and consistent. Modelica uses the keyword **partial** to indicate that a model is incomplete and uninstantiable.

A resistor has all properties described by the `TwoPin` model. Therefore, the TwoPin model can be reused in defining a resistor model:

```
model Resistor "Ideal electrical resistor"
  extends TwoPin;
  parameter Real r (unit="Ohm") "Resistance";
equation
  R * i = v;
end Resistor;
```

The **extends** clause specifies that `TwoPin` is the parent class of resistor. By inheriting a class, it inherits all attributes and equations from the parent class. Modelica supports multiple inheritance.

Variables declared with prefix **parameter** are called parameter. The value of a parameters is constant during a simulation run, but it can be changed between runs. This makes it possible for a user to change the behavior of a model without recompiling it.

1.2.7 Modeling Dynamics

Dynamic systems have behavior which evolves as a function of time. Modelica has a unique predefined independent variable `time`. All Modelica variables are implicitly signals: their value varies with `time`.

The output of a sine-wave voltage source is a function of time. The following definition of voltage source shows that.

```
model VsourceAC "sin-wave voltage source"
  extends TwoPin;
  parameter Voltage VA = 110 "Amplitude";
  parameter Real f (unit="Hz") = 50 "Frequency";
  constant Real PI = 3.14159265;
equation
  v = VA*sin(2*PI*f*time);
end VsourceAC;
```

Also, Modelica uses the predefined operator **der** to represent the time derivative. It occurs in the model definition of a Capacitor.

```
model Capacitor "Ideal electrical capacitor"
  extends TwoPin;
  parameter Real c (unit="F") "Capacitance";
equation
  c * der(v) = i;
end Capacitor;
```

where `der(v)` means the time derivative of `v`.

1.3 Current Tools

There already exist some excellent commercial tools for Modelica. *Dymola*, an integrated modeling and simulation tool developed by *Dynasim* (`http://www.dynasim.se`), has a Modelica translator which is able to perform all necessary symbolic transformations for large systems (more than 100 000 equations) as well as for real time applications. It includes a graphical editor for model editing and browsing, and a simulation environment. It also provides convenient interfaces to Matlab and

the popular block diagram simulator Simulink. For example, a Modelica model can be transformed into a SIMULINK S-function which can be simulated in Simulink as an input/output block.

Another commercial tool for Modelica is *MathModelica* developed by MathCore (`http://www.mathcore.com`). It provides a Modelica simulation environment which is closely integrated into Mathematica (`http://www.wolfram.com`). The tight integration with Mathematica also makes it possible to perform complex analysis tasks, advanced scripting, and other technical computations on models and simulation results. MathModelica has a graphical editor for model editing and browsing. The kernel of MathModelica is similar to that of Dymola because internally, the Dymola symbolic and simulation engine is used for the formula manipulation and for the simulations.

The *Open Source Modelica* is a tool developed by PELAB, Linköping University (`http://www.ida.liu.se/labs/pelab/modelica`). It is to create a complete Modelica modeling, compilation and simulation environment based on free software distributed in source code form intended for research purposes. The *Open Source Modelica* tool contains two major modules, *Modeq* and *ModSimPack*. Modeq is a translator which translates Modelica source model into flat Modelica, while ModSimPack is a translator that translates flat Modelica model to C/C++ code.

1.4 Thesis Objectives

In the long term, we are interested in developing an integrated modeling and simulation environment for Modelica, as well as using Modelica as a meta-modeling language with AToM^3, a tool for multi-formalism and meta-modeling under development at the Modeling, Simulation, and Design Lab (MDSL) in the School of computer Science of McGill University (`http://moncs.cs.mcgill.ca/MSDL/research/projects/AToM3/`). A Modelica compiler is required as the kernel for this future environment. With limited resources, our research currently focuses on a subset of the Modelica language, and on continuous systems. We name this subset μModelica, where μ stands for *mini*, *meta-modeling*, *multi-formalism*, and *MSDL*.

The main objective of this thesis is to build an efficient research prototype compiler for μModelica. More specifically, the first prototype of the μModelica compiler was designed to provide an interative environment that supports the real essence of the language—non-causal modeling. It not only performs semantic analysis, but also carries out some computer algrebra optimization techniques in terms of formula manipulation.

Also, this thesis is to provide a relatively complete specification of the semantics of μModelica, and to sumerize and propose our studies of some language features and formula manipulation techniques as future work.

2

The Overall Architecture

Given that our aim is to build an open *source research* prototype compiler for Modelica, rapid prototyping and portability are the main concerns in choosing the implementation language. Python is an interpreted, dynamically type-checked object-oriented programming language. Like Java, the Python implementation is portable across many platforms. But compared to JAVA, it is better-suited for rapid software prototyping. Python also supports the seamless integration of code developed in statically type-checked language. This "extension" allows the gradual replacement of performance-critical parts of the prototype. It also allows gluing of libraries (e.g., numerical code written in Fortran). In addition, in order to build an integrated modeling and simulation environment with a graphical user interface, the μModelica compiler will be embedded into AToM^3, which was implemented in Python. With the above-mentioned advantages of Python, and for consistency in our future tool, we chose to implement the μModelica compiler in Python.

As a research prototype compiler, this project currently only focus on a subset of the Modelica language. But this subset covers the real essence of Modelica—non-causal modeling. The μModelica compiler is able to resolve class inheritance and translates input models into flat Modelica, and performs symbolic transformations on the DAEs. Support for advanced and complex language constructs is left as future work. Following is a list of language features that are not yet supported in the μModelica compiler:

- the causal modeling constructs, i.e. algorithm statement, function call, and **block** class
- arrays and matrices
- element redeclaration
- the import statement
- external function call
- the `within` construct
- hybrid system modeling, e.g. conditional equations, `when` equations
- and more ...

2.1 The Big Picture

Figure 2.1 shows a high-level view of the μModelica compiler. The compiler consists of three modules, the *Front End*, the *Back End*, and the *Code Generator*. The Front End takes Modelica source code as input, performs lexical and semantic analysis, and generates a flat Modelica model, which is in essence a set of DAEs. The flat Modelica model is then passed to the Back End, where formula manipulation is done. The Code Generator finally generates input for the Octave simulator. More details on Octave will be discussed later. Also, for the purpose of testing and debugging, pretty printers are employed to dump Modelica code from internal representation at

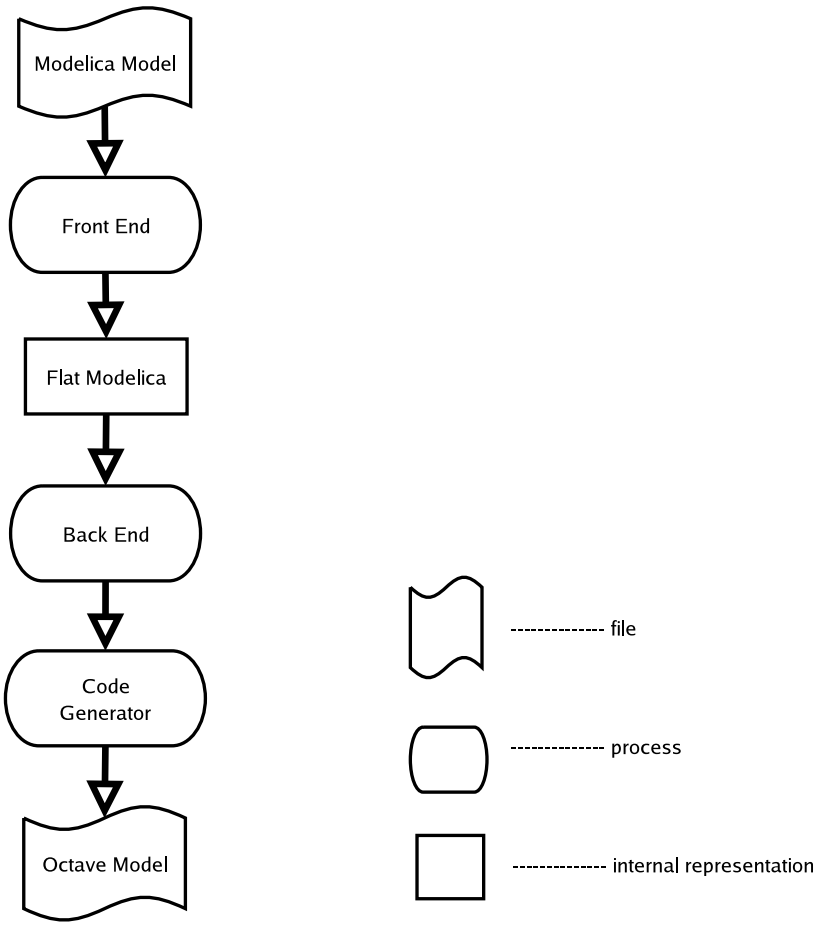

Figure 2.1: The Overview of μModelica

different phases. By comparing this Modelica code with the input model, we can verify that no information is lost during each transformation step. It is important to note that debugging output of *each* step of model compilation is in the form of a *valid Modelica model* (which is accepted by the compiler).

2.2 The Front End

Figure 2.2 is a detailed view of the Front End. The Front End is made up of the following components:

- A Linux executable (implemented in C++) that takes a Modelica model as input, performs parsing, and generates an XML representation of the parse tree.
- A graph kernel called *pyGK* (implemented in Python), which reloads the XML parse tree into a pyGK graph representation.
- The ASTBuilder, which converts the parse tree into an abstract syntax tree.
- A semantic analyzer, which translates the original Modelica model into flat Modelica.

Thanks to PELAB at the Department of Computer and Information Science, Linköping University, who have kindly offered us the lexer and parser of Modelica from their Open Source Modelica project, we were able to save a lot of time in implementing the parser. The PELAB parser was

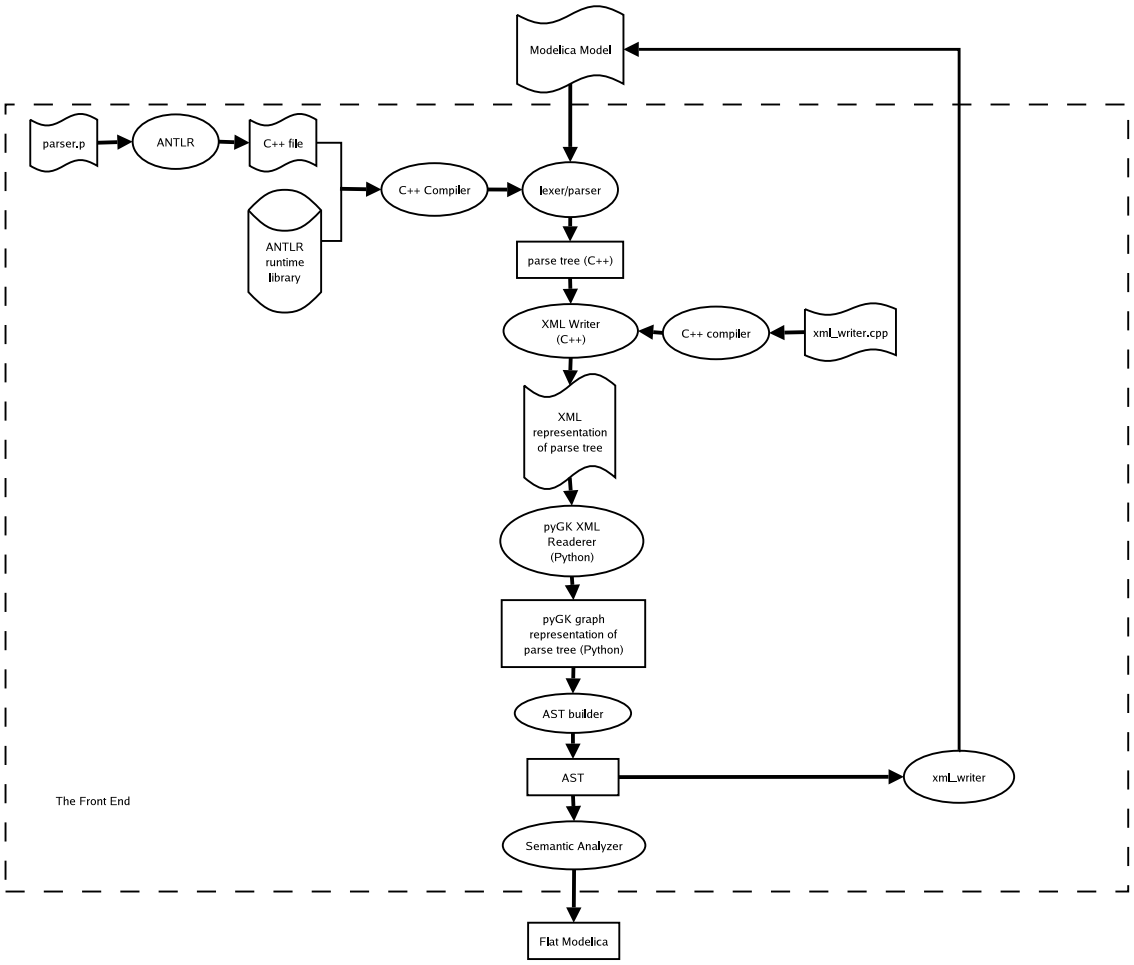

Figure 2.2: The Front End of μModelica

developed using the tool ANTLR under Windows. We recompiled it under Linux, and generated a C++ parser.

This parser accepts Modelica models and generates parse trees. Implementing the parser and the following parts of the compiler in different languages raised a problem: how to pass the parse tree to Python? One solution is to “extend” Python with the parser. Another solution is to use an external data description, through which the two different languages can communicate and exchange data. This provides a stricter separation but is less efficient. *XML* is a mark up language for describing structured data. It provides a mechanism to identify structure in data. In our design, we chose to write out to file an XML description of the parse tree, and then reload this information and transform it into a Python parse tree. So long as the XML representation is well-defined, no information will be lost during this transformation. The XML representation and transformation process will be discussed in section 3.2.

A parse tree represents the *concrete syntax* of a model. It is more desirable to have an internal representation of the *abstract syntax* of a model. The abstract syntax of a Modelica model is defined in terms of language constructs, such as *class, element, declaration, statement, and expression* etc. This representation is independent of the source syntax of a Modelica model being compiled. A parse tree is transformed to an *abstract syntax tree* (AST) by the `ASTBuilder`. In order to verify that the AST transformation is correct and complete, Modelica source code is produced from the

AST. Verification of correctness can be done by comparing the Modelica model thus produced to the Modelica input model.

Semantic analysis is carried out in the Front End. It includes scoping analysis, name lookup, expanding inheritance, flattening structured components and coupled models, and type checking. Finally, the Front End generates a flat Modelica model, where all structured components are flattened down to basic components, and connections are replaced by regular equations. Such a flat model is a system of DAEs.

2.3 The Back End

Figure 2.3 gives a detailed view of the Back End.

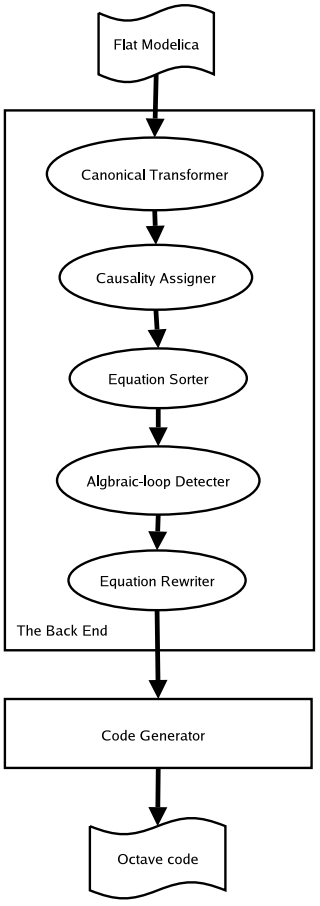

Figure 2.3: The Back End of μModelica

Automated formula manipulation is significant to non-causal modeling, which is characterized by a set of implicit equations (DAEs). A simple approach to solving for the various unknowns in the set of equations is to call a DAE solver, such as DASSL. However, the solution will be far more efficient if a causal representation can be found, i.e. computational causality is assigned and equations are sorted in an appropriate computation order. In many cases it is possible to transform a non-causal set of equations into a causal one. Even though this transformation process trades off compile time efficiency, it gains in simulation run-time. Since the number of simulation run is usually much greater than the number of compilations, it is certainly worths to try these transformations.

Causality assignment, along with sorting of equations and algebraic loop detection, is implemented in the μModelica compiler. Causality here means *computational causality*, not *physical causality*. Causality assignment determines that which variable is to be computed in each equation. The computational causality can be found in terms of some graph algorithms, which will be discussed later.

Once a computation of causality is found, the equations are then sorted into a correct computation order based on their computational dependency. However, if there exist *algebraic loops*, causality assignment and sorting are not sufficient. The equations involved in an algebraic loop are mutually dependent. Sorting is not able to give a correct computation order. These equations must be identified and be solved separately. Therefore, sorting is always followed by the detection of algebraic loops.

In order to provide an optimal internal representation of equations, a canonical representation of equations and corresponding transformation rules are defined in [26]. A subset of these transformation rules are implemented in the μModelica compiler.

2.4 The Code Generator

Finally, input for the *Octave* code is generated. Octave (a GNU Matlab clone), provides a high-level language for numerical computation. Octave has extensive tools for solving linear algebra problems, nonlinear equations, and integrating ordinary differential and differential-algebraic equations. The Octave model is executed by an Octave interpreter. The computation result can be visualized through GNU plot. We generate Octave input for convenience. The generated output is similar to Matlab M-files. To maximize efficiency, Simulink S-function will be generated in the future.

3

The Front End

The Front End performs parsing and semantic analysis. The Modelica semantics is defined in terms of a set of rules for translating classes (including inheritance and modification), instances, and connections into flat Modelica, which is a flat set of constants, variables, and equations. For example, the following Modelica model `Main`

```
class A
  Real a1, a2;
equation
  a1 * 2 = a2;
end A;

class B
  Real b1, b2;
equation
  b1 ^ 2 = b2;
end B;

model Main
  A a;
  B b;
equation
  a.a1 = b.b1;
  b.b2 = 4;
end Main;
```

will be translated into the following flat Modelica:

```
model Main "flat"
  Real a_a1;
  Real a_a2;
  Real b_b1;
  Real b_b2;
equation
  a_a1 * 2 = a_a2;
  b_b1 ^ 2 = b_b2;
  a_a1 = b_b1;
  b_b2 = 4;
end Main;
```

where all structured components, such as `a, b`, are flattened to basic components, such as `a_a1, b_b1` etc. The equation part is the mathematical description of the model.

3.1 The Parser

As mentioned earlier, the parser was developed by PELAB at Linköpings University for its *Open Source Modelica* project. It was implemented in ANTLR. ANTLR, Another Tool for Language Recognition (`http://http://www.antlr.org`), is a language tool that provides a framework for constructing recognizers, compilers, and translators from grammatical descriptions. It is able to generate parsers in Java, C#, or C++, but not Python. The PELAB Modelica parser uses C++. We have compiled it under Linux using the *gcc* compiler.

The parser takes Modelica source code as input, and generates a parse tree. A parse tree is made up of nodes, which are defined and provided by ANTLR's AST factory. Each tree node corresponds to a token. It records the *type* and *value* of a token, and keeps a reference to its first child and a reference to its next sibling. Figure 3.1 is the class diagram view of a parse tree node.

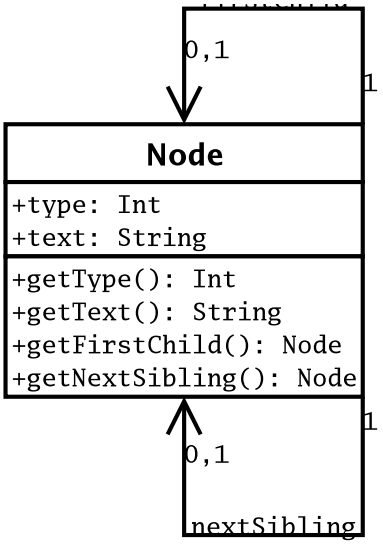

Figure 3.1: Class Diagram of Parse Tree Node

Given the following Modelica model as input,

```
// Example 1
model A
  Real a;
end A;
```

a parse tree pretty printer dumps out the following parse tree:

```
+-117 : STORED_DEFINITION {
| +-95 : CLASS_DEFINITION {
| | +-36 : model
| | +-84 : A
| | +-98 : DECLARATION {
```

```
| | | +-84 : Real
| | | +-84 : a
| | | +-97 : COMMENT
| | }
| }
}
```

The higher abstraction view of this parse tree is shown in Figure 3.2.

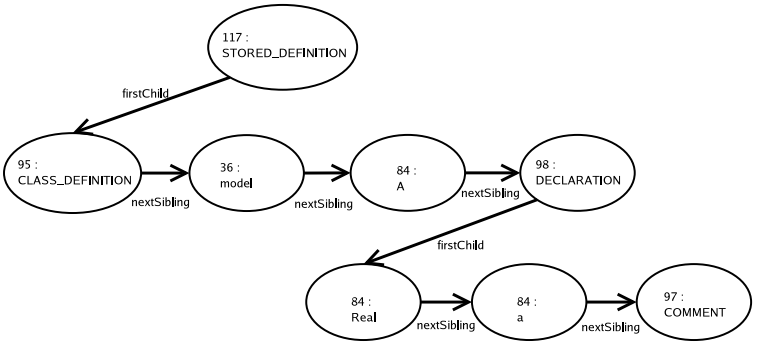

Figure 3.2: A High-level Abstraction View of a Parse Tree

3.2 XML Representation

In order to make a parse tree accessible to our compiler written in Python, an XML representation of the parse tree is defined. The parser together with an XML writer, writes out a parse tree in such an XML representation. This XML file is then reloaded by an XML reader (written in Python).

Again, for rapid prototyping, a graph kernel called *pyGK* was employed to define the XML representation and to reload the XML file as a graph representation of a parse tree. pyGK was developed in Python, as the graph kernel for AToM^3, by Marc Provost at the MDSL, School of computer Science at McGill University (`http://moncs.cs.mcgill.ca/people/mprovost/`). It provides an XML representation for graphs, and an XML Reader and an XML Writer. Note that using pyGK also opens up the possibility of applying graph transformations to the parse tree.

In the Front End, there is an XML writer (in C++) which writes out the parse tree into an XML file in pyGK format. The XML file is then reloaded by the XML reader in pyGK into the data structure defined in pyGK—a graph representation of the parse tree in Python. Figure 3.3 shows this process.

3.2.1 Representation

A pyGK graph XML representation conceptually contains two parts, a list of nodes, and a list of edges. This is different from the parse tree representation, which is based on adjacent nodes. The parser takes care of this conversion when it writes out XML. Adjacencies between nodes are explicitly written out as edges. Example 1 in section 3.1 is represented in XML as follow:

```
<?xml version="1.0"?>
<!DOCTYPE agl SYSTEM "http://agl.dtd">
<!-- GraphElements -->
<agl>
```

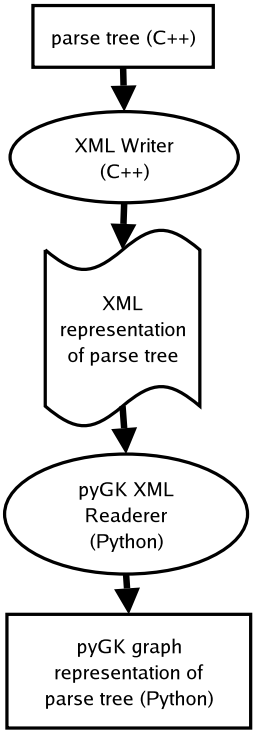

Figure 3.3: Parse Tree in C++ to pyGK Graph

```
<graph id="AST" type="AST">
    <symb id="0" type="SymbolTable">
        <map key="text">
            <string id="" type="String" value="STORED_DEFINITION"/>
        </map>
        <map key="type">
            <int id="" type="Int" value="117"/>
        </map>
    </symb>
    <symb id="1" type="SymbolTable">
        <map key="text">
            <string id="" type="String" value="CLASS_DEFINITION"/>
        </map>
        <map key="type">
            <int id="" type="Int" value="95"/>
        </map>
    </symb>
    <symb id="2" type="SymbolTable">
        <map key="text">
            <string id="" type="String" value="model"/>
        </map>
        <map key="type">
            <int id="" type="Int" value="36"/>
        </map>
    </symb>
```

```
    <symb id="3" type="SymbolTable">
        <map key="text">
            <string id="" type="String" value="A"/>
        </map>
        <map key="type">
            <int id="" type="Int" value="84"/>
        </map>
    </symb>
    <symb id="4" type="SymbolTable">
        <map key="text">
            <string id="" type="String" value="DECLARATION"/>
        </map>
        <map key="type">
            <int id="" type="Int" value="98"/>
        </map>
    </symb>
    <symb id="5" type="SymbolTable">
        <map key="text">
            <string id="" type="String" value="Real"/>
        </map>
        <map key="type">
            <int id="" type="Int" value="84"/>
        </map>
    </symb>
    <symb id="6" type="SymbolTable">
        <map key="text">
            <string id="" type="String" value="a"/>
        </map>
        <map key="type">
            <int id="" type="Int" value="84"/>
        </map>
    </symb>
    <symb id="7" type="SymbolTable">
        <map key="text">
            <string id="" type="String" value="COMMENT"/>
        </map>
        <map key="type">
            <int id="" type="Int" value="97"/>
        </map>
    </symb>
    <edge from="0" to="1" fromOrd="0" toOrd="0"/>
    <edge from="1" to="2" fromOrd="0" toOrd="0"/>
    <edge from="2" to="3" fromOrd="1" toOrd="0"/>
    <edge from="3" to="4" fromOrd="1" toOrd="0"/>
    <edge from="4" to="5" fromOrd="0" toOrd="0"/>
    <edge from="5" to="6" fromOrd="1" toOrd="0"/>
    <edge from="6" to="7" fromOrd="1" toOrd="0"/>
</graph>
```

```
</agl>
```

A parse tree node is represented as a *SymbolTable*. Node attributes, i.e. `type` and `text`, are stored in the SymbolTable as entries. A SymbolTable entry is a mapping from *key* to *value*. For example, a parse tree node with the token `Real` is converted to

```
<symb id="5" type="SymbolTable">
    <map key="text">
        <string id="" type="String" value="Real"/>
    </map>
    <map key="type">
        <int id="" type="Int" value="84"/>
    </map>
</symb>
```

where `type` is mapped to `84` (integer value of IDENTIFIER), and `text` is mapped to `Real`.

In order to represent edges, each graph node is assigned a globally unique ID. For example, node `Real` and node `a` in the declaration `Real a` are assigned `5` and `6`, respectively. In the parse tree, node `a` is an adjacent node of node `Real`. But in the XML representation, such an adjacency is explicitly described by an edge

```
<edge from="5" to="6" fromOrd="1" toOrd="0"/>
```

This representation denotes an outgoing edge from node `5`, which is `Real`, to node `6`, which is `a`. The field *fromOrd* indicates whether this edge points to a child or sibling (`0` means child, `1` means sibling).

A parse tree is converted to a directed pyGK graph by means of the transformation denoted above. Even though pyGK represents a parse tree differently, the conceptual structure is maintained.

3.2.2 Implementation Issues

The main implementation issue in the parse tree transformation via XML representation is process management. At the top level of the computer, the data flow control is implemented in Python. But at the very beginning of the flow, the parser program (compiled C++) needs to be executed to generate an XML file, which will then be reloaded by a Python script. Generating and reloading of the XML file have to be synchronized. Therefore, the problem is how to create and manage a new process in which the parser is executed.

Python has a module called `os`, which provides access to operating system functionality. It includes a series of functions for process management. One of them is the `spawn*(mode, file, ...)` function, which executes the program `file` in a new process. The asterisk means that it has variants. We use one of the variants, `spawnlp(mode, path, file, ...)`, to manage the parsing and XML generating process. The *l* variants are designed to be used in the case that the parameters of program `file` are fixed, while the *p* variants will use `path` to locate the program `file`. For example,

```
os.spawnlp(os.P_WAIT, parserPath, './xmlDumper', arg1, arg2)
```

creates a new process which executes the program `xmlDumper` at location `parserPath`, with `arg1` and `arg2` as parameters. Under the `os.P_WAIT` mode, the main process will be temporarily suspended till the new process exits. This mechanism guarantees that XML reloading will not happen until the file has been generated.

3.3 Abstract Syntax

Each node in a syntax tree must encode information to indicate the kind of the node. There are two ways to encode this information: the homogeneous approach and the heterogeneous approach. The homogeneous approach uses a single class type together with numerous token types to represent tree nodes. A syntax tree with homogeneous nodes can be seen as a *parse tree*. The heterogeneous approach uses different classes to represent different kinds of tree nodes. A syntax tree constructed in this way, such as *Abstract Syntax Tree*, is a heterogeneous structure.

The syntax tree generated by the PELAB parser is a homogeneous parse tree. A detailed description of the representation of tree nodes has been given in section 3.1. A parse tree represents the concrete syntax of the corresponding program, and it is source code dependent. In order to perform semantic analysis more efficiently, a program is typically represented in terms of an *abstract syntax tree* (AST) internally in a compiler. The *abstract syntax tree*, along with the *Visitor* design pattern, is one of the most important patterns in compiler design and implementation. Note that the homogeneous parse tree representation does not support the visitor pattern. More details on the visitor pattern are given in section 3.4.3.

3.3.1 Design

An abstract syntax of Modelica is defined. Even though this project currently only works on a subset of the language, the full Modelica syntax is supported (implemented), from parsing to abstract syntax tree construction. The abstract syntax is specified in terms of the Modelica language constructs. According to these constructs, the abstract syntax is divided into four packages: *Definition*, *Component*, *EquationPart*, and *Expression*. Also, a package called *Scope* is defined to support scoping analysis. Figure 3.4 is the UML diagram of the abstract syntax at package level.

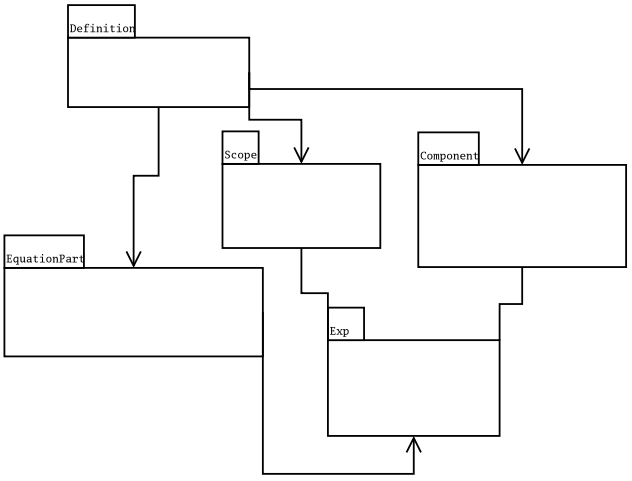

Figure 3.4: Packages in Abstract Syntax

Figure 3.5 shows the classes defined within package *Definition*. *ClassFile* represents the highest level construct—a Modelica file. A ClassFile consists of *class definitions*, which in turn are made up of *elements*. In Modelica, element refers to class definitions, extends-clauses, and component declared in a class. A more detailed description of these language constructs can be found in the Modelica Syntax defined in [5].

Figure 3.6, Figure 3.7, Figure 3.8, and Figure 3.9, describe the detailed definition of package *Com-*

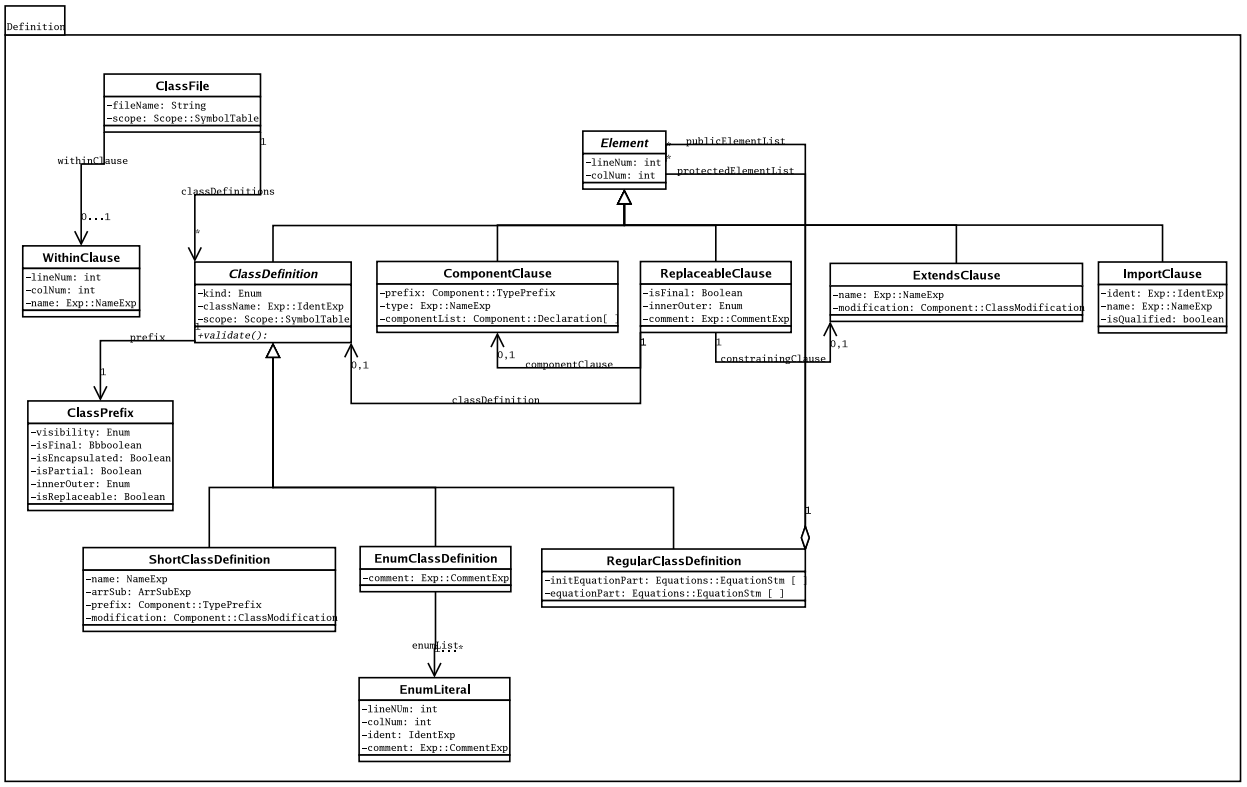

Figure 3.5: Package Definition

ponent, EquationPart, Expressions, and *Scope*, respectively. Each of these classes corresponds to a specific Modelica language construct defined by the Modelica grammar in [5]. See `Appendix` for a full description of the Modelica grammar.

3.3.2 Transformation

The transformation from parse tree to AST is implemented in a one-pass tree traversal. Each structural sub-tree is transformed to a corresponding abstract syntax construct. Parentheses, which are syntactically significant to expressions, no longer appear explicitly in ASTs. The AST structure contains the structural information of an expression.

3.3.3 Test of Correctness

A series of transformations are executed to convert a Modelica model into an AST. This AST has to retain all the information from the source model. To verify this requirement, a pretty printer is implemented to dump out syntactically correct Modelica. That is, given an AST, print out its corresponding Modelica model. If the Modelica printout is the same as its original input, then it proves that all the transformations are correct. In the μModelica compiler, a user can choose whether or not to dump Modelica.

3.4 Scoping and Name Lookup

As in traditional programming languages, a name in a Modelica model has a *scope* in which it is visible. The scope of a name is the region of the code where the name has a meaning corresponding to its intended use [15]. Scoping analysis is characterized by the introduction and maintenance of

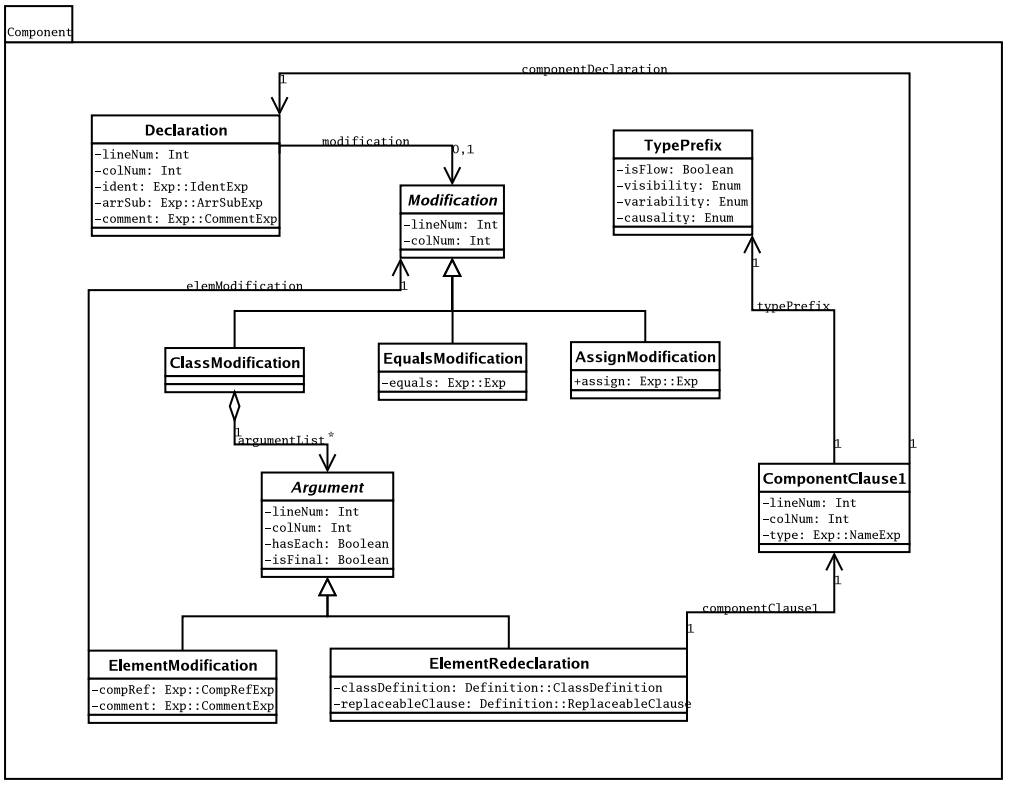

Figure 3.6: Package Component

symbol tables, which store mappings of identifiers to their types and definitions. As class definitions and declarations are processed, *bindings* from identifiers to their meanings are added to the symbol tables. When identifiers are used, they are looked up in the symbol tables and *bound*.

The following sections explains some Modelica language constructs, which are significant in understanding the scoping rules of Modelica. The data structure for scoping analysis is then presented.

3.4.1 Semantics

Variable Declaration

In Modelica, class instances are created via variable declarations. A declaration states the type and other properties of a variable. A declaration in Modelica has the following form:

```
[prefixes] type-specifier component-list
```

where

- `prefixes` specifies accessibility, variability, and data flow.
- `type-specifier` specifies the *type* of a variable.
- `component-list` a list of component declaration. A component declaration is an *identifier*, optionally followed by an array dimension descriptor and/or *modification*.

For example, the following declaration

```
parameter Real a, b;
```

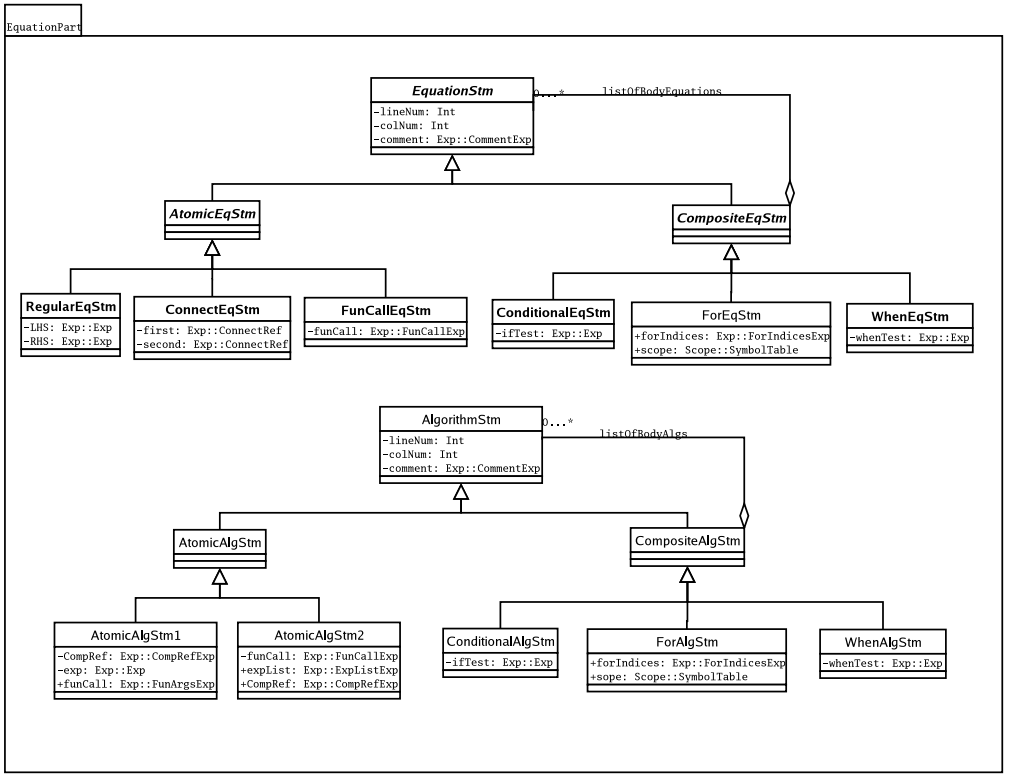

Figure 3.7: Package Equations

declares two *parameters*, *a* and *b*, of type *Real*. It is equivalent to the following:

```
parameter Real a;
parameter Real b;
```

Local Class Definition

Modelica supports local class definition. Local classes can be defined nested inside a class. The number of levels of nesting is unlimited. The following example shows that class `B` is locally defined within class `A`:

```
class A
  class B
    Real x;
  end B;

  class C
    B b1;
  end C;

  B b2;
end A;
```

A local class definition is accessible in the class where it is defined, and from within all local nested classes. In this example, local class definition `B` can be accessed from anywhere in `A` and `C`.

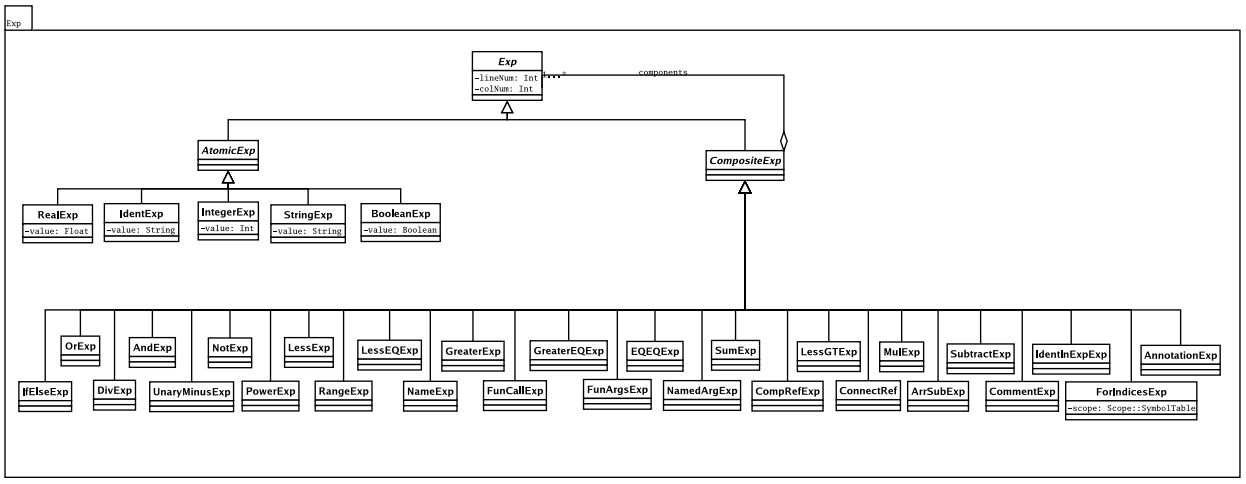

Figure 3.8: Package Expressions

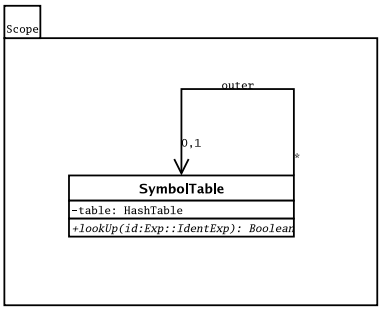

Figure 3.9: Package Scope

The Concept of Parents

The classes lexically enclosing an element form an ordered set of parents. A class defined inside another class definition (the parent) precedes its enclosing class definition in this set [5]. For example:

```
class A
  ...
end A;

class B
  Real x;

  class C
    Real y;
  end C;
end B;
```

There is an unnamed parent at the top-level, which encloses all class definitions. In the example, the parent of class `A` and `B` is that unnamed parent. The ordered set of parents of class `C` is `[B, unnamed]`, and the one of variable `y` is `[C, B, unnamed]`.

Encapsulated Class

Class encapsulation is defined by the class prefix `encapsulated`. It is a mechanism for scoping control. Elements declared in a parent class are not accessible from within an encapsulated class unless they are explicitly imported. In the previous example, if class `C` is defined as encapsulated, `b1` can not be declared as an instance of `B`. The following model shows the correct use of encapsulation in class `D`.

```
class A
  class B
    Real x;
  end B;

  encapsulated class C
    B b1; // error
  end C;

  encapsulated class D
    import A.B;
    B b1; // correct
  end D;

  B b2;
end A;
```

Use-Before-Declare

The current Modelica language (version 2.0) allows *use-before-declare* (UBD). This language feature provides better support for graphical user environments, because the order of declaration is not determined when components are graphically created.

In some programming languages, the name of a class definition can be used before the class is defined. In Modelica, not only class definitions, but also variables can be used before they are declared. Below is an example demonstrating this feature.

```
class A
  B b;
  class B
    Real x(start=y);
    parameter y = 2;
  end B;
end A;
```

In class `A`, name `B` is used before it is defined. Also, within class `B`, variable `y` is referenced before it is declared. Both these declarations are legal in Modelica.

Illegal Referencing of Declared Variables

It has been shown in the previous section that a declared class is accessible from within local nested classes. But this rule does not apply to declared variables (except for constants). In Modelica, it is illegal to reference *variables* or *parameters* declared in parent classes. Referencing declared *constants* in parent classes is allowed. For example:

```
class A
  class B
    Real x;
  end B;

  class C
    B b1; // referencing class definition in parent class is legal
  end C;

  Real y;
  constant Real z = 10;

  class D
    Real d1(start=z); // referencing declared constants is also legal
    Real d2 = y; // error: referencing declared variables is illegal
  end D;

  B b2;
end A;
```

For-loop

The following clause

```
for IDENT1 in expression1, IDENT2 in expression2 ... loop
  loop body
end for
```

defines a for-loop in Modelica. A for-loop introduces an additional lexical scope. Variables declared in a for-loop (called *iteration variables*) are visible only within the body of the for-loop. The following example clearly shows how the scope of an iteration variable is just the body of the for-loop.

```
class B
  constant Integer j=4;
  Real x[j];
equation
  for j in 1:j loop // The loop variable j takes the values 1,2,3,4
    x[j]=j; // Uses the loop variable j
  end for;
end B;
```

In the for-loop index, the first `j` is implicitly declared as an iteration variable, while the second `j` refers to the constant integer declared in class `B`.

Short Class Definition

In addition to the regular class definition form, a class can also be defined in terms of the following short form

```
class IDENT1 = IDENT2 class_modification;
```

which is identical to

```
class IDENT1
  extends IDENT2 class_modification;
end IDENT1;
```

except that short class definition does *not introduce an additional lexical scope for modifiers*. The following example taken from [5] demonstrates the difference:

```
model Resistor
  parameter Real R;
  ...
end Resistor;

model A
  parameter Real R;
  model Load=Resistor(R=R);
  // this is correct because the R in Resistor is set to R from model A

  model LoadError
    extends Resistor(R=R);
    // this gives the singular equation R=R, since the right-hand side
    // R is looked up in LoadError and found in its base-class Resistor
  end LoadError;
  ...
end A;
```

This is an exception to the lookup rules: modifiers in short classes are looked up in the immediately enclosing scope.

Scope of Predefined Names

User-defined classes are built from predefined types, functions etc. The predefined types in Modelica are `Real, Integer, Boolean`, and `String`. Modelica also has predefined functions such as `der()`, and `sin()`. All these predefined names are accessible from anywhere within a program, including encapsulated classes. Thus, the scope of predefined names is *global*.

Duplicate Declarations

In Modelica, an element name is unique in each lexical scope. Therefore, duplicate element names are not allowed in a class. The name of a declared element must be different from all other declared elements in that class. For example, both of the following classes are illegal:

```
class A
  Real a;
  Integer a; // Error: duplicate variable name
end A;

class B
  Real X;
```

```
  class X  // Error: X has been declared as Real in this scope
    ...
  end X;
end B;
```

Introduction of the Keyword self

In order to print out intermediate representation of Modelica models as valid Modelica, we introduce the keyword **self** in μModelica. The meaning of **self** is equivalent to that of **self** in Python. That is, it refers to the object that holds the scope within which it is used. For example, the following model

```
model A
  Real a=3.0;
  Real b;
equation
  a*b=15;
end A;
```

can be rewritten as

```
model A
  Real a(self=3.0);
  Real b;
equation
  self.a * self.b = 15;
end A;
```

where the first **self** refers to `a`, the second and the third **self** refers to model `A`.

Scope Rules

From the previous examples, scope rules in Modelica can be summarized as follow:

- The scope of a top-level class definition is the entire program, except for those classes which are encapsulated.
- The scope of a local class definition or a declared constant covers the whole class where it is declared, including all local nested classes that are not encapsulated.
- The scope of a declared variable or parameter is the enclosing class, excluding local nested classes.
- The scope of an iteration variable is the body of the for-loop where it is implicitly declared.

Static Name Lookup

When a class is being instantiated, names used within that class are looked up. These names include type specifiers, variables, and functions. According to the nested scope rules, Modelica uses hierarchical lookup of names. The lookup starts searching from the scope where it is used, then searches in the ordered set of enclosing scopes until a match is found or an enclosing class is defined as encapsulated.

A name in Modelica can either be a simple name (without dot referencing, e.g., A), or a composite name (composed using dot notation, e.g., A.B.C). The lookup procedure of a simple name is as follows:

- If a name is inside a for-loop or inside the body of a reduction expression, it is looked up starting in the for-loop scope which contains the implicitly declared iteration variables.
- A name is then looked up sequentially in each member of the ordered set of parents, in the built-in scope and in the predefined scope, until a match is found or a parent is encapsulated.
- If the name is not found declared in the previous two steps, the lookup continues in the global scope which contains predefined names.

The lookup in each scope is performed as follow:

- Among declared components and local class definitions, including those inherited from base classes.
- Among the import names of qualified import statements.
- Among the public elements of packages imported via unqualified import statements.

For a composite name of the form `A.B.C`:

- The first identifier is looked up as a simple name.
- If the first identifier denotes a declaration, the rest of the name, e.g, `B.C`, is looked up among the declared components of the definition of the declaration.
- If the fist identifier denotes a class, the rest of the name is looked up among the declared name elements of the class. If the class does not satisfy the requirements of a package, the lookup is restricted to encapsulated elements only. The following example shows this restriction:

```
package P1
  constant Real a=3.0;
  class B1
    Real b1=a;
  end B1;
end P1;

class P2
  Real a;
  encapsulated class B2
    Real b2;
  end B2;
  class B3
    Real b3;
  end B3;
end P2;

model M
  P1.B1 x1; // This is legal since P1 satisfies
            // the requirements of a package.
  P2.B2 x2; // This is also allowed because B2 is an encapsulated
```

```
                // class even though P2 is not a package.
  P2.B3 x3; // This is illegal because neither P2 is a
                // package nor B3 is encapsulated.
end M;
```

3.4.2 Design and Implementation

The basic structure of semantic analysis is the *symbol table*. It is a tabulation of the collection of declarations and contextual information, which is convenient for symbol lookup. Each symbol in the table is bound to the meaning as it is declared. This section presents the design and implementation of scoping analysis and name lookup in the μModelica compiler.

The Data Structure

The analysis in section 3.4 shows that scopes usually correspond to class definitions. Each class definition introduces a new lexical scope. If a scope is conceptually seen as a node, the structure of scopes in a program can then be seen as a tree.

In the AST design, a *Scope* node has been inserted at each structure level which introduces a lexical scope. These include `ClassFile`, `ClassDefinition`, `ForEqStm`, `ForAlgStm`, and `ForFunArgExp` (reduction expression). A scope node is actually a symbol table and has two attributes: *table* and *outer*. *table* is a dictionary (in Python) which maps identifiers to their "meanings" (`TableEntry`), and *outer* is a reference to the scope node of its outer scope. With this *outer* attribute, scope nodes are connected as a tree, with the scope node at the `ClassFile` level as the root. Figure 3.10 is the class diagram of the scope package.

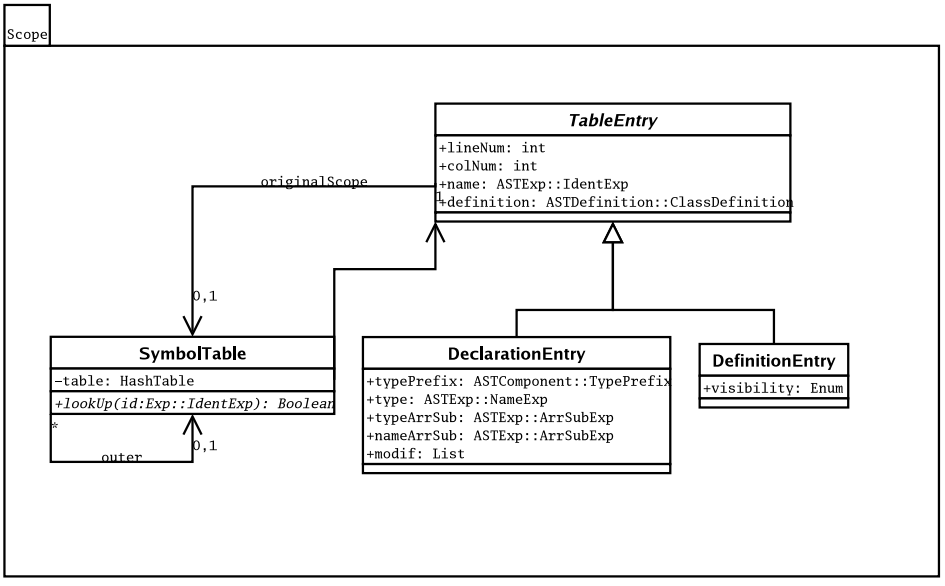

Figure 3.10: Detailed Package Scope

A name (identifier) is bound to certain "meanings" in symbol tables. The "meanings" of a name is represented in its corresponding `TableEntry`, which contains information such as prefix, type, definition, value, etc. All the information of declarations in an AST is moved to symbol tables, which enables efficient symbol lookup.

Multiple Passes

As mentioned earlier, Modelica (version 2.0) allows **use-before-declare** (UBD). *Multiple passes* are required to support UBD in implementing scoping analysis and name lookup. In the μModelica compiler, these tasks are executed through three passes. Each pass carries out a specific task, which is done via a *visitor*.

During the first pass, the visitor collects all declared elements, i.e., declared components and local classes, in the AST. A new `TableEntry` is created for each declared name. There are two types of `TableEntry`, `DeclarationEntry` and `DefinitionEntry`. The class diagram in Figure 3.10 shows the relationship. More specifically, a `DeclarationEntry` is created to store all the information of a declared component, while a `DefinitionEntry` is created for a declared class. A `DefinitionEntry` mainly keeps a reference to the class definition node in the AST. These created entries are added to symbol tables in corresponding scopes. Each symbol table then contains all declaration information in its own scope after the first pass completes.

In a lexical scope, whenever a declaration with duplicate name is detected, an exception is thrown.

Figure 3.11 illustrates the AST and scope nodes structure of the following Modelica model:

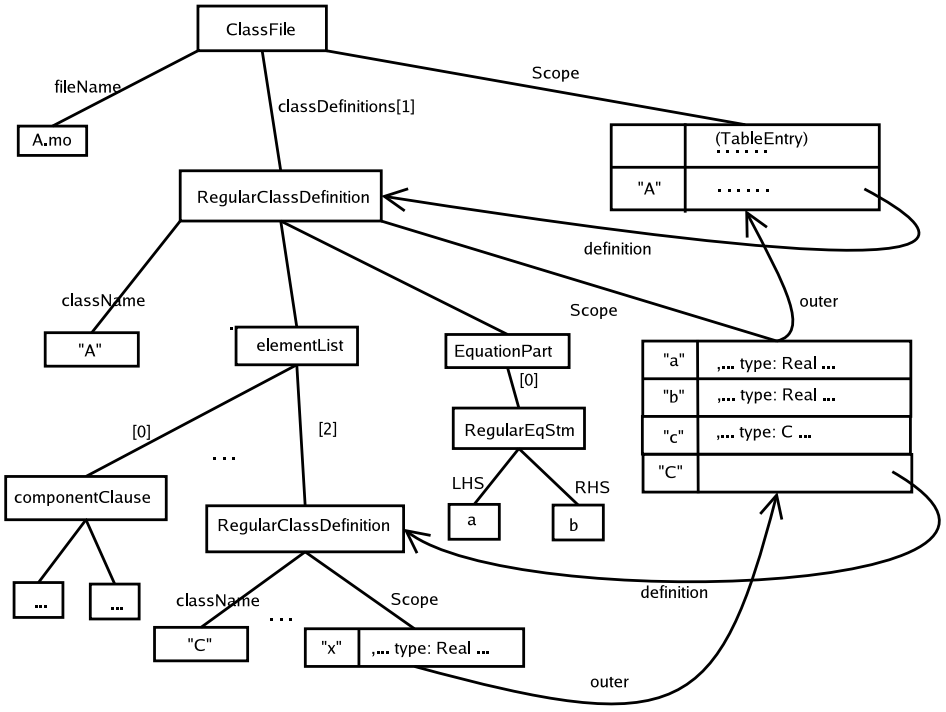

Figure 3.11: AST after the first pass

```
// Assume that this Modelica model is stored in file A.mo
class A
  Real a, b;

  C c;

  class C
    Real x;
```

```
  end C;

equation
  a = b;
end A;
```

The second pass is to perform type specifiers lookup. For example, in the following declaration

```
Integer a;
```

`Integer` is the type specifier. A visitor iterates over all `DeclarationEntry`s in the AST. The type specifier in each of these symbol table entries is looked up. The lookup algorithm is given in section 3.4.1. If the type specifier is found, a reference to the definition that type in the AST is created in that `DeclarationEntry`, otherwise an exception is thrown. Figure 3.12 shows the change in the AST of the previous example after the second pass.

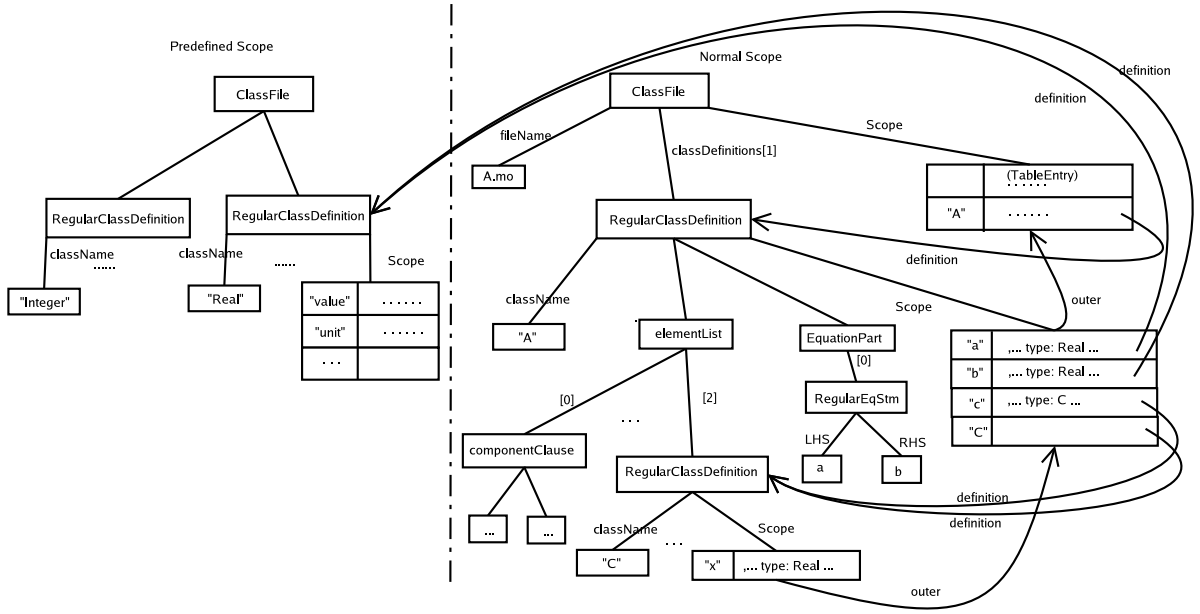

Figure 3.12: AST after the second pass

Used names are looked up during the third pass. Names are used in modifications and equations. For example, in the following piece of code

```
model Example
  ...
  parameter Real a;
  Real b;
  Resistor R1(r=2*a);
  Resistor R2;
  ...
equation
  b = a^2;
  connect(R1.p, R2.n);
```

```
  ...
end Example;
```

`R1` is declared as a `Resistor`, together with *modification* $r = 2 * a$ to its attribute `r`. In this modification, names `r` and `a` are used. Also, in the equation part, names `b`, `a`, `R1.p`, and `R2.n` are referred to. To verify that the use of these names is legal, they must be found declared in that lexical scope. The lookup algorithm is also the same as the one given in section 3.4.1, except that the left hand side of an element modification, e.g., `r` in $r = 2 * a$, is looked up in its definition scope. For example, `r` is looked up in the scope where class `Resistor` is defined.

If use of a name is verified legal, a reference to the `DeclarationEntry` of this name in a symbol table is created. Otherwise, an exception is thrown. Figure 3.13 shows the same AST as the one in Figure 3.11, after the third pass.

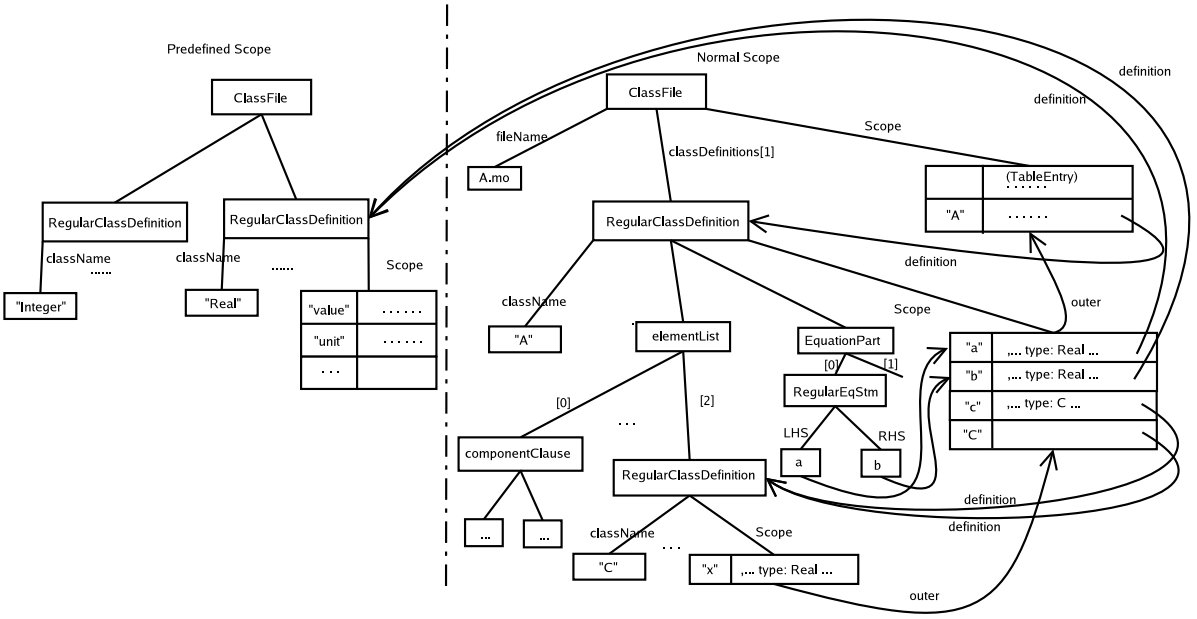

Figure 3.13: AST after the third pass

3.4.3 The Visitor Design Pattern

In order to give a clean design and an easy-to-understand implementation, we use the *visitor* design pattern. *Visitor* enables the complete separation of data and the operations to be performed on the data. In other words, one can define a new operation without changing the classes of the elements on which it operates.

Overview of the Visitor Pattern

Programs are usually represented as *Abstract Syntax Trees* (AST) internally in a compiler. An *abstract syntax tree* is a structure which consists of different types of elements (nodes). The semantic analysis will need to perform operations, such as name lookup, type checking, etc., on ASTs. Moreover, we might define operations on ASTs for pretty-printing, program restructuring, and code generation. Most of these operations might treat different kinds of tree nodes differently. For example, type-checking for sum expressions is different from type-checking for function calls.

If all these operations are coded as methods inside various class definitions of AST nodes, it leads to a system that is hard to understand, maintain, and change.

Another design scheme is to separate the data and the operations performed on it. More specifically, we can place related operations, e.g., type-checking operations for various sorts of nodes, into a separate object, namely, a *Visitor*. The visitor is then passed to elements of the AST as the AST is traversed. Each AST node has an *accept* method with the *visitor* object as argument. The *accept* method in each AST node in turn invokes the method in the *visitor* that is specifically defined for this kind of nodes. This method invocation includes the AST node itself as an argument. It is the *visitor* that executes the operation for AST nodes. This technique is called *double-dispatch* because the operation that gets executed depends on both the type of the *visitor* and the type of the element.

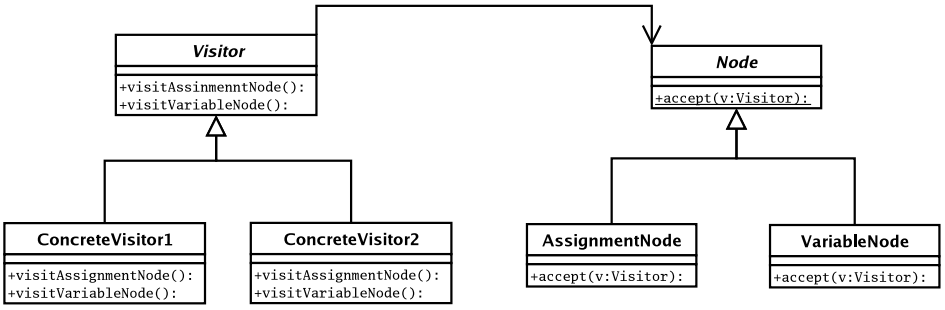

Figure 3.14: The Visitor Pattern

Figure 3.14 is the UML class diagram of the *Visitor Pattern*. There are two class hierarchies in this pattern: the class hierarchy of nodes (data) and the class hierarchy of visitors (operations performed on data). Each concrete visitor encodes an operation to be performed on various kinds of nodes. One can add a new operation in the compiler by creating a new subclass in the visitor class hierarchy. As long as the grammar of a language does not change, new functionality of the compiler can be augmented simply by adding new concrete visitor classes.

Implementation Issues

As it is mentioned earlier, the underlying philosophy of the visitor pattern is *double-dispatch*. *Double-dispatch* simply means that the operation that gets executed depends on the kind of request and the types of two receivers: the visitor's and the element's. The *accept* method is a *double-dispatch* operation. The following block of code (Python style) is a sample *accept* method:

```
class VariableNode:

  ...

  def accept(self, v):
    v.visitVariableNode(self)
```

The *accept* method at every element is invoked during the object structure traversal. But the real operation is not performed within the *accept* method. Instead, it sends a request to the visitor to execute the operation on the element being visited. Therefore, the operation that gets executed depends on both the type of Visitor and the type of Element it visits. This is the key to the visitor pattern.

The second implementation issue is object structure traversal. A visitor is supposed to visit each element in the structure. The problem is, *who is responsible for traversing the structure*? Or in other words, *how does the visitor get there*?

In fact, we can put the structure traversal code either in the object structure itself, or in the visitor. If we put responsibility for traversing the structure in the object structure itself, we only need to write the traversal code once in the object structure. But each time a visitor traverses the structure, it has to follow the same traversal algorithm. In our implementation, the visitors are responsible for traversing the AST. Each concrete visitor has its own code for each aggregate concrete element. This is because in our compiler, the traversal algorithm for some visitors are different, even though it ends up duplicating code. For example, a name lookup visitor only needs to traverse the equation part in an AST.

The Design

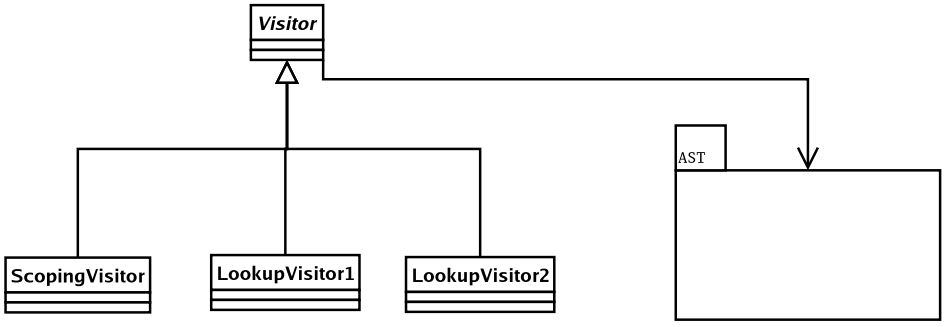

Figure 3.15: The Name Lookup Visitor

Figure 3.15 is a simplified UML class diagram of the design of the visitors that perform scoping analysis and name lookup. A class hierarchy of visitors is defined to support multiple concrete visitors. The parent class `Visitor` of all visitors of an AST is an abstract class. The parent `Visitor` declares an operation (method) for each AST construct class it visits. In every concrete visitor, there is a corresponding implementation for each of these methods. All these methods are not shown in the class diagram because the number of AST constructs is relatively large.

3.5 Expanding Inheritance

Modelica supports class inheritance, a key feature of object-oriented language. An existing class, called *superclass* or *base class*, can be extended to define a more specialized class, which inherits the properties and behavior of the base class. The specialized class is called *subclass* or *derived class*. It is defined in terms of the *extends* clause

```
extends name [class_modification]
```

For example, the following piece of code

```
partial model TwoPin "Superclass of elements with 2 electrical pins"
  Pin p, n;
  Voltage v;
  Current i;
equation
```

```
  v = p.v - n.v;
  0 = p.i + n.i;
  i = p.i;
end TwoPin;

model Resistor "Ideal electrical resistor"
  extends TwoPin;
  parameter Real r (unit="Ohm") "Resistance";
equation
  R * i = v;
end Resistor;
```

defines `Resistor` as a subclass of TwoPin.

3.5.1 Semantics

As mentioned earlier, the goal of the Front End is to translate original Modelica model into flat Modelica, where all declared components are flattened down to predefined types. In class inheritance, properties and behavior are inherited in the form of component declarations, and equations. Derived classes need to be expanded before the model is translated into flat Modelica. In fact, all data and equations in base classes are *copied* to the derived class. The following example shows how extend clauses are expanded.

```
package P
  constant Real PI=3.14;
  class A
    Real a1, a2;
  equation
    a1*a2=1.0;
  end A;

  class B
    A a(a2=PI);
    Real b;
  equation
    a.a2 * b^2 = 10.0;
  end B;
end P;

class C
  class C1
    Real c11;
  end C1;
end C;

model M
  extends C;
  extends P.B;
  C1 x;
```

```
end M;

// Expanding the extends clauses in model M leads to the following
// expanded version of M:
model M
  // inherited from C
  class C1
    Real c11;
  end C1;

  // inherited from P.B
  constant Real PI=3.14;
  A a(a2=PI);
  Real b;

  C1 x;
equation
  a.a2 * b^2 = 10.0;
end M;
```

In this example, model `M` is a derived class of both class `C` and class `B` in package `P`. In the process of expanding the extends clauses, all declared elements, including variables and local class definitions, and equations in base classes, are copied to the derived class `M`. In addition to that, referenced constant declarations in the enclosing scope by base classes, e.g., `constant Real PI=3.14` in this example, are also copied to the derived class.

It is possible that an inherited component from a base class has the same name as a locally declared component. As it is mentioned in section 3.4, two declared components in a lexical scope are not allowed to have the same name. But under certain conditions, the inherited declaration and the local one can be merged into a single declaration. These conditions are:

- the two declarations must have the same type;
- they must have the same protection level, i.e., they must be both public elements or protected elements;
- they must have the same type prefix, e.g., they must be both declared as continuous variable, or both declared as discrete variables, or both declared as parameters, etc.

Satisfying these conditions means that the two declarations are identical. One of them is ignored. Otherwise it is illegal, which will cause an exception in the compiler. For example,

```
class A
  Real a, b;
end A;

class B
  extends A;
  Real a;     // This is legal.
  Integer b;  // This is illegal since the type is different to
              // to the one declared in class A.
end B;
```

This rule also applies to inheritance of equations. If an inherited equation is syntactically identical to a locally declared one in the derived class, one of the equations is discarded. For example, in the expanded class `F`, only one equation is kept.

```
class E
  Real e1, e2;
equation
  e1+e2=1;
end E;

class F
  extends E;
equation
  e1+e2=1;
end F;

// The expanded version of F
class F
  Real e1;
  Real e2;
equation
  e1+e2=1;
end F;
```

3.5.2 Multiple Inheritance

Modelica supports multiple inheritance, i.e., more than one *extends clause* is allowed. In some cases this might lead to the problem of inheriting the same element or equation twice through other intermediate inheritances. For example,

```
class Person
  String name;
  Integer dateOfBirth;
end Person;

class Student
  extends Person;
  Integer stuID;
end Student;

class Employee
  extends Person;
  Integer emplID;
end Employee;

class StudentEmployee
  extends Student;
  extends Employee;
end StudentEmployee;
```

Class `Student` and Employee are derived classes of base class `Person`, while class `StudentEmployee` is a derived class of both class `Student` and Employee. In the process of expanding class `StudentEmployee`, the declarations `String name` and `Integer dateOfBirth` are inherited twice, via the two *extends* clauses. This is an example of *repeated* inheritance. This problem can be solved easily by applying the rule stated in the previous section. According to that rule, only one of the identical components is kept. Class `StudentEmployee` is expanded as:

```
class StudentEmployee
  String name;
  Integer dateOfBirth;
  Integer stuID;
  Integer emplID;
end StudentEmployee;
```

3.5.3 Modification of the Extends Clause

An extends clause may carry class modifications which modify the value of attributes in base classes. For example,

```
class A
  Real a1=2.0;
  Real a2=3.0;
end A;

class B
  extends A(a2=5.0);
  Real b;
end B;
```

When an extends clause is expanded, the modification is also applied to corresponding elements in the base class. Finally, element modifications are merged, that is, outer modification overrides inner modification. In the μModelica compiler, each declaration symbol table entry carries a list of modifications. In the process of expanding inheritance, modification is appended to the copy of corresponding elements, which are then inserted into the derived class. The following is the pretty printed version of the expanded version of class `B` by the μModelica compiler:

```
// Expanded version of B
class B
  Real a1=2.0;
  Real a2(self=3.0, self=5.0);
  Real b;
end B;
```

3.5.4 Short Class Definition as Class Inheritance

A short class definition is a concise way of defining new classes based on inheritance. As described in section 3.4.1, the short form class definition

```
class IDENT1 = IDENT2 class_modification;
```

is usually identical to

```
class IDENT1
  extends IDENT2 class_modification;
end IDENT1;
```

which is a longer form based on the extends clause. Actually in the μModelica compiler, short class definitions are transformed to the above longer form, which are then expanded. All semantic analysis performed on short classes are identical to normal classes, except that modifiers in short classes are looked up in the immediate enclosing scope.

3.5.5 The Process of Expanding Inheritance

The process of expanding inheritance consists of the following steps:

- The name of the base class is looked up;
- If the base class contains unexpanded extends clauses; recursively expand all extends clauses in the base class;
- Copy all declared elements and equations from the base class to the derived class;
- Resolve class modification and apply the modification to corresponding inherited components.

3.5.6 Implementation Issues

Component Deep Copy

In the process of expanding class inheritance, the modification needs to be resolved and be applied to the declarations copied from the base class. Since modification creates a variant of the original definition, shared objects are no longer sufficient to carry modification of multiple instances. Therefore, true copies of declarations are needed to store the information of modification.

These true copies can be created via a deep copy operation. Python's deep copy operation creates copies which are unnecessarily "too deep" with respect to what is required here. A user-defined deep copy operation is needed to make sure that these copies are created only as deep as needed, but not more. In the case that modification is absent, it suffices to keep a reference to the original element object when it is copied from the base class.

Detecting Cyclic Dependency of Inheritance

Cyclic dependency of inheritance must be detected when extends clauses are expanded (to avoid infinite expansion). The following example shows a cyclic dependency of inheritance between class `A, B` and `C`:

```
class A
  extends B;
  Real a;
end A;

class B
  extends C;
end B;
```

```
class C;
  extends A(a=1.0);
  Real c;
end C;
```

Such a dependency can be detected by introducing a list of visited base class when inheritance is recursively expanded. Suppose class `A` is the first to be expanded in this example, `A` is visited and is then added to the list of visited classes, as [`A`]. Since `A` extends class `B` and `B` extends class `C`, `B` and `C` are recursively visited and are added to the list, giving [`A, B, C`]. Finally when the extends clause in class `C` is expanded, its base class `A` is again visited and will be added to the list [`A, B, C`], in which `A` already exists. Duplicate occurrences of the same class in the list means there exists a cyclic inheritance dependency. An exception is raised when such a dependency is detected.

3.5.7 Order of Expanding Inheritance and Name Lookup

In order to guarantee that elements can be used before they are declared, expanding inheritance and name lookup are executed in the following order in the μModelica compiler:

1. The first pass: class definitions and component declarations are added into symbol tables;
2. The second pass: class inheritances are resolved, that is, inherited elements are copied from based classes to the derived class;
3. The third pass: *type specifiers* are looked up;
4. The fourth pass: uses of names in modifications and in the equation part are looked up.

3.6 Flattening

The goal of semantic analysis is to translate a Modelica model to Flat Modelica, which consists of basic components and DAEs. For example, the following model `M`

```
class A
  Real a1;
  Integer a2=1;
end A;

class B
  Real b1(unit="N");
  Real b2=2.0;
end B;

model M
  extends A;
  B b(b1=1.5, b2=3.0);
end M;
```

is translated to the following flat form of Modelica by the front end:

```
model M
  Real  a1;
```

```
  Integer  a2;
  Real  b_b2;
  Real  b_b1;
equation
  a2=1;
  b_b2=3.0;
  b_b1=1.5;
end M;
```

In the flattened model, all class attributes are declared in terms of predefined types, and modifications are merged and turned into equations.

After expanding extends clauses, a class contains attributes which can either be basic components or composite components. Actually composite components are built up of basic components. The purpose of flattening is to expand the class structure into a flat form, i.e., all class attributes are declared as predefined types. The key issues in the flattening process are:

1. Component instantiation;
2. Flattening of composite components and merging of modifications;
3. Generation of connection equations.

3.6.1 Component Instantiation

In the internal representation of a Modelica model, instead of creating a concrete object instance for each declaration, a symbol table entry just keeps a reference to its class definition, and carries all the original information of modifications. Figure 3.16 shows the internal representation of the previous example.

This scheme helps save memory space, and it works well until modifications are resolved. Class modification creates a variant of the original class definition. The data structure has to be augmented to hold concrete instances for these variants.

An instance stores the modified data and behavior of a component. It consists of a symbol table, a list of initial equations, and a list of equations. The symbol table contains class attributes copied from the original class definition, as well as the modification to that component. The equation part is also copied from class definition to instance.

A class modification contains element modifications. Each element modification is looked up and is applied to the found element in the concrete instance. For space efficiency, no instance will be created for declarations which contain no modification. It is sufficient to keep references to their definitions.

In the μModelica compiler, components are instantiated as follow:

```
instantiateComp(comp):
  if comp has no modification:
    if the type of comp is not a predefined type:
      for each subcomponent(i) in comp:
        instantiateComp(i)
  else:
    inst=createInstance(comp)
    merge modification to inst
```

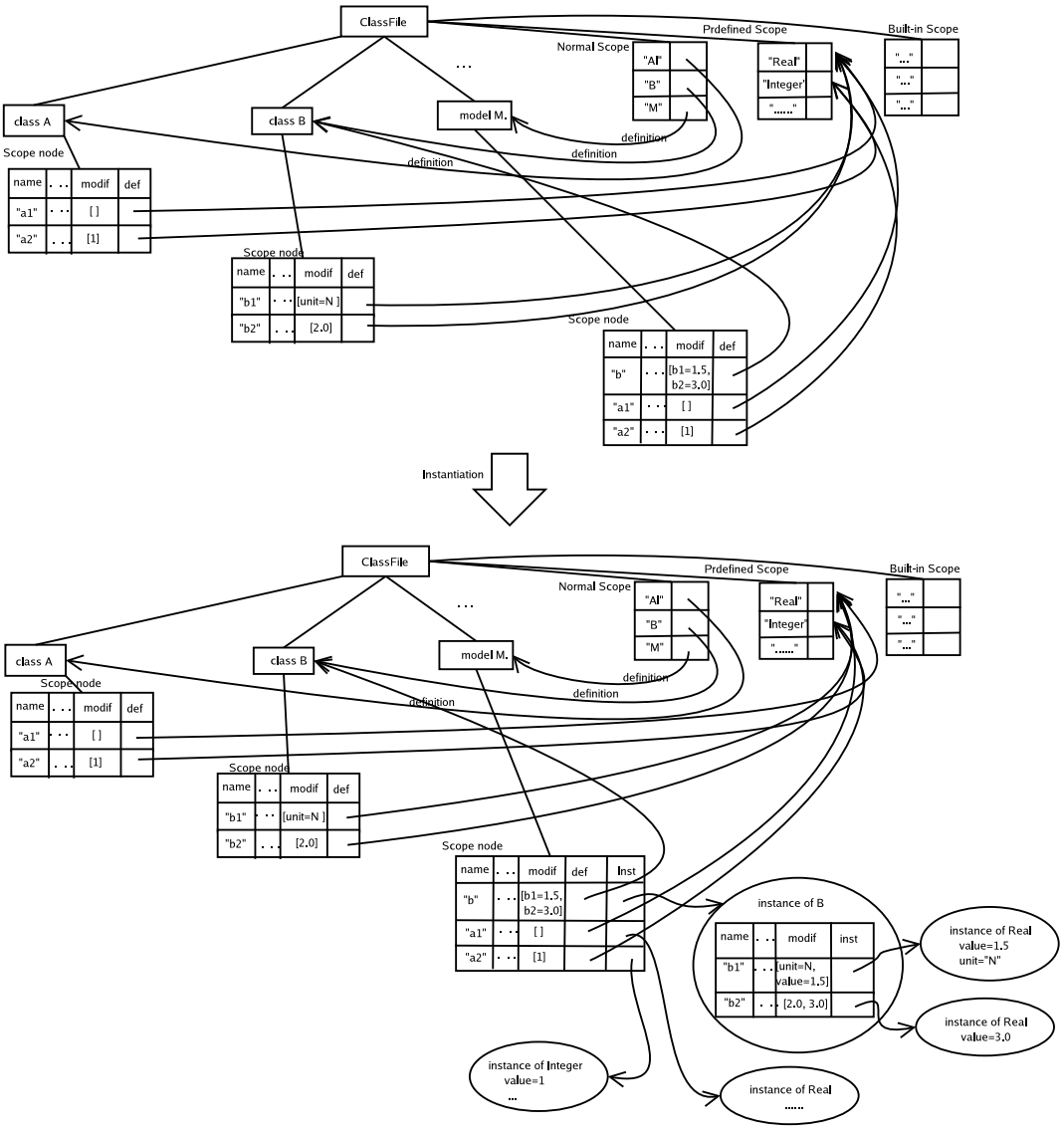

Figure 3.16: Component Instantiation

```
if the type of comp is not a predefined type:
  for each subcomponent(j) in inst:
    instantiateComp(i)
```

Figure 3.16 also shows the internal representation after components are instantiated.

3.6.2 Flattening of Composite Components

A flat Modelica model is a system of equations with all variables declared in predefined types, i.e., a declared variable must be one of the type *Real*, *Integer, Boolean*, or *String*. In the process of flattening, all structured components, i.e., composite components, are expanded to a set of basic components.

Resolving Variable Names

When composite components are replaced by basic components, names from different lexical scopes are inserted into the same lexical scope. This will easily lead to a problem: some declared variables might have the same name in the flattened model, which is in conflict with the rule that no duplicate declarations of the same name are allowed. Therefore, the basic components in a flattened model must be renamed such that each of them has a unique name in its lexical scope.

Renaming of basic components in a flattened model can be achieved by combining the names of a basic component and the names of all its parents. This scheme guarantees that each combined name is unique because each name and each of the name in its order set of parents is unique in its own lexical scope. For example, in model `M`

```
model M
  extends A;
  B b(b1=1.5, b2=3.0);
end M;
```

the composite component `b` is replaced by

```
  Real b_b1;
  Real b_b2;
```

where *b_b*1 and *b_b*2 are the combinations of `b` and the name of basic components in base class `B`, namely `b1, b2`, respectively.

As mentioned earlier, the μModelica compiler is able to dump out intermediate representations as valid Modelica models. For example, it can dump out a flat Modelica as a valid Modelica model. In some other Modelica tools, *dot* notation is used in combining names in flat Modelica. For example, `b_b1` and `b_b2` will become *b.b*1 and *b.b*2, respectively. But according to the grammar of Modelica, a declared identifier is not allowed to contain *dot*. Among those special characters, *underscore* is the only one that is valid in an identifier. Therefore, it is used to make names unique in the μModelica compiler.

Merge of Modifications

Modifications can be applied not only to declared components or local classes, but also to extends clauses. Therefore, it is possible that a component can be modified multiple times through these *nested* modifications. For example,

```
class A
  Real a1;
  Real a2=2.0;
end A;

class M
  A a(a1=1.0, a2=3.0);
end M;
```

there are two modifiers $a2 = 2.0$, and $a2 = 3.0$, applied to `a2`. But finally a variable can only take one value. This means that nested modifications must be merged.

In Modelica, outer modification takes precedence over inner modifications. $a2 = 3.0$ is an outer modification compared to $a2 = 2.0$ in the previous example. In the μModelica compiler, each declared variable carries a list of modifications. The priority of modifications is encoded in the list. After composite components have been flattened to basic components, modifications for each of these basic components are merged. In the following example,

```
class A
  Real a1;
  Integer a2=1;
end A;

class B
  Real b1(unit="N");
  Real b2=2.0;
end B;

model M
  extends A;
  B b(b1=1.5, b2=3.0);
end M;
```

merging modifications in the flattened model `M` gives

```
// flattened version of M with modifications merged
model M
  Real  a1;
  Integer  a2=1;
  Real  b_b1(unit="N")=1.5;
  Real  b_b2=3.0;
end M;
```

Other than representing a single value, a basic component, i.e., instance of a predefined type, has other attributes which describe some properties of a physical quantity. For example, type `Real` is defined in Modelica syntax as follow

```
type Real
  RealType value;
  parameter StringType quantity ="";
  parameter StringType unit ="";
  parameter StringType displayUnit ="";
  parameter RealType min =-Inf; // Inf denotes a large value
  parameter RealType max =Inf;
  parameter RealType start =0;
  parameter BooleanType fixed =false; // "true" for parameter/constant
  parameter BooleanType enable =true;
  parameter RealType nominal;
  parameter StateSelect stateSelect = StateSelect.default;
equation
  assert(value >= min and value <= max, "Variable value out of limit");
```

```
  assert(nominal >= min and nominal <= max, "Nominal value out of limit");
end Real;
```

Modifications may apply to some of these attributes. In the μModelica compiler, an environment is created for each basic component, which contains mappings from attribute to value. Figure 3.17 shows the environment for

```
Real b_b1(unit="N")=1.5;
```

Some information encapsulated in a basic component might not be so important to the simulation back end, such as attribute ”unit”. But it will be important in the future if we are to implement unit-based type checking. For the sake of integrity, this information is kept in the environment.

ATTRIBUTE	VALUE
value	1.5
quantity	None
unit	None
displayunit	None
min	-Inf
max	Inf
start	0
fixed	false
enable	true
norminal	None
stateSelect	StateDelect.edfault

Figure 3.17: Environment of Basic Component Real

Modification Equation

Modifications in Modelica are finally turned into equations. In the simulation back end, we are currently only concerned with the *value* attribute of each basic component. Other attributes, such as *unit*, are ignored. Therefore, only modifications to the *value* field are turned into equations. Other modifications are stored in the environments. For example, the modification in basic component

```
Real b_b1(unit="N")=1.5;
```

is turned into

$$b\_b1 = 1.5$$

Model `M` in the previous example is then finally flattened to

```
model M
  Real a1;
  Integer a2;
  Real b_b1;
```

```
  Real b_b2;
equation
  a2 = 1;
  b_b1 = 1.5;
  b_b2 = 3.0;
end M;
```

The information of modification `unit="N"` is not lost. It is maintained in the environment of *b_b*1. It is typically used to generate code usable in an integrated modeling and simulation environment such as WEST [27].

3.6.3 Generation of Connection Equations

In Modelica, components may be coupled by connections, whose semantics are given by equations. A connection is introduced by the following `connect` statement:

```
connect (connector1, connector2)
```

where `connector1` and `connector2` are two references to connectors, each of which is either a component of the same class or an element of one of its component. `connector1` and `connector2` must be *type equivalent*. Two types T1 and T2 are *equivalent* if:

- T1 and T2 denotes the same primitive type, i.e., one of `RealType`, `IntegerType`, `StringType`, `BooleanType`, or `EnumType`;
- T1 and T2 are classes containing the same public declaration elements (according to their names) and each of these elements in T1 is type equivalent to the corresponding one in T2.

Connection statements are converted into normal equations, which are called *connection equations*. There are two different forms of connection equation generated, for *flow* and *non-flow* variables, respectively. The main tasks in the process of generating connection equations are:

- Building *connection sets* from connection statements;
- Generating connection equations for the complete model.

Connection Sets

In Modelica, multiple connections can be made to a *single* connector. For example, in the simple electronic circuit in section 1.2:

```
model Circuit
  Resistor R1(r=1);
  Resistor R2(r=1);
  Capacitor C(c=1);
  VsourceAC AC;
  Ground G;
equation
  connect (AC.p, R1.p);
  connect (R1.n, R2.p);
  connect (R2.n, C.p);
  connect (C.n, AC.n);
  connect (AC.n, G.p);
end circuit
```

connector `AC.n` is connected to both connector `C.n` and `G.p`. Multiple connections to a single connector can be seen as a node where all involved connectors are inter-connected. Through (flow) variables from each connector are summed to zero at such a node.

Connection sets are needed to detect if there exist connectors that have multiple connections. Connection sets are first built at the level of connectors, i.e., *connector connection sets*. Then these connector connection sets are expanded to *primitive connection sets*.

- A *connector connection set* is a set of connectors connected by means of *connect clauses*. All connectors in such a set are type equivalent.

The algorithm for building connector connection sets in the μModelica compiler is as follows:

1. Create an empty list L to store connection sets;
2. For each connect statement, create a connection set containing its two arguments, and append this set to list L. For example, the connection set of `connect (C.n, AC.n)` is $[C.n, AC.n]$;
3. Merge connection sets. That is, if any connector in a connection set is a member of other connection sets, all the corresponding sets are merged. For example, the connections set of connect statement `connect (AC.n, G.p)`, $[AC.n, G.p]$, is merged with $[C.n, AC.n]$, resulting in $[C.n, AC.n, G.p]$.

All connectors in a connection set are type equivalent, i.e., they have the same public attributes (with the same names and types). If a connector type contains structured components, these structured component are expanded into basic components, i.e., predefined types. Common basic component of members in a connector connection set form a *primitive connection set*:

- A set of variables having the same name and the same Modelica predefined type.
- A primitive connection set may only contain flow variables or non-flow variables.

For example, the connector class of the electronic circuit example in section 1.2 is defined as:

```
connector Pin
  Voltage v;
  flow Current i;
end pin;
```

The generated primitive connection sets from the connector connection set $[C.n, AC.n, G.p]$ are:

- Non-flow variables: $[C.n.v, AC.n.v, G.p.v]$
- Flow variables: $[C.n.i, AC.n.i, G.p.i]$

Connection Equations for Non-flow Variables

Equations generated from primitive connection sets of non-flow variables have the following form:

$$a_1 = a_2 = ... = a_n$$

which is equivalent to:

$$\begin{aligned} a_1 &= a_2 \\ a_1 &= a_3 \\ &... \\ a_1 &= a_n \end{aligned}$$

Connection Equations for Flow Variables

Equations generated from primitive connection sets of flow variables have the following form:

$$b_1 + b_2 + ... + (-b_k) + ... + b_n = 0$$

The sign of variable b_i is +1 if the connector is an *inside* connector, and -1 if the connector is a *outside* connector. Inside and outside connectors are defined as follows:

- In an element instance M, a connector component of M is called an *outside* connector with respect to M.
- A connector component that is hierarchically inside M is called an *inside* connector with respect to M.

For example, all members of connector connection set $[C.n, AC.n, G.p]$ are *inside* connectors of model `Circuit`. Therefore, the generated equation for the primitive connection set of flow variables $[C.n.i, AC.n.i, G.p.i]$ is:

$$G\_p\_i + AC\_n\_i + C\_n\_i = 0.0$$

3.7 Type Checking

After all the above-mentioned semantic analysis are executed, a Modelica model is translated into *flat Modelica*. In the μModelica compiler, the textual representation of flat Modelica is still valid Modelica model. For example, the flat Modelica representation of the circuit example is as follow:

```
model Circuit
  flow Real  R1_n_i;;
  Real  R2_v;;
  parameter Real  R1_r;
  Real  G_p_v;;
  Real  C_n_v;;
  Real  R1_p_v;;
  flow Real  R1_p_i;;
  flow Real  G_p_i;;
  Real  R2_i;;
  flow Real  AC_n_i;;
  parameter Real  R2_r;
  Real  R1_n_v;;
  flow Real  R2_n_i;;
  parameter Real  C_c;
  flow Real  R2_p_i;;
  parameter Real  AC_VA;
  flow Real  AC_p_i;;
  Real  C_i;;
  flow Real  C_p_i;;
  constant Real  AC_PI;
  Real  R1_v;;
```

```
    Real  AC_v;;
    Real  AC_i;;
    Real  C_p_v;;
    Real  R1_i;;
    Real  C_v;;
    Real  AC_p_v;;
    Real  AC_n_v;;
    Real  R2_p_v;;
    flow Real  C_n_i;;
    Real  R2_n_v;;
    parameter Real  AC_f;
  equation
    C_c*der(C_v)=C_i;
    C_v=C_p_v-C_n_v;
    0=C_p_i+C_n_i;
    C_i=C_p_i;
    R1_r*R1_i=R1_v;
    R1_v=R1_p_v-R1_n_v;
    0=R1_p_i+R1_n_i;
    R1_i=R1_p_i;
    G_p_v=0;
    R2_r*R2_i=R2_v;
    R2_v=R2_p_v-R2_n_v;
    0=R2_p_i+R2_n_i;
    R2_i=R2_p_i;
    AC_v=AC_VA*sin(2*AC_PI*time);
    AC_v=AC_p_v-AC_n_v;
    0=AC_p_i+AC_n_i;
    AC_i=AC_p_i;
    AC_p_i+R1_p_i=0.0;
    AC_p_v=R1_p_v;
    R1_n_i+R2_p_i=0.0;
    R1_n_v=R2_p_v;
    R2_n_i+C_p_i=0.0;
    R2_n_v=C_p_v;
    G_p_i+AC_n_i+C_n_i=0.0;
    G_p_v=AC_n_v;
    G_p_v=C_n_v;
  end Circuit;
```

As in other programming languages, the type of a construct in Modelica need to match what is expected in its usage context. For example, the - (minus) operator expects two operands of type `Integer` or `Real`. Therefore, the expression of $a - b$ is a type error if any of `a` or `b` is neither an integer nor real number.

This leads to the next step to be performed in the compiler: type checking. The main tasks of type checking are:

- Verify that each construct is type correct.

- Type coercion, which changes the type of one expression to another.

3.7.1 Basic Types

As it is mentioned earlier in previous sections, the predefined types in Modelica are built over the built-in types, i.e., `RealType, IntegerType, StringType`, and `BooleanType`. By default, the name of a variable of predefined types refers to its *value* attribute. For example, equation

$$a + b = c$$

means

$$a.value + b.value = c.value$$

The basic types in Modelica are `RealType`, `IntegerType`, `StringType`, `BooleanType`, `TypeError`, and `Void`. The basic type `void` represents the empty set and allows an equation to be checked. `TypeError` indicates a construct has type errors.

3.7.2 Type Coercion

In some cases, an operator allows its operands to have different types. For example, the expression $a + b$ is legal in Modelica, where a is of type `Real` and b is of type `Integer`. Since the machine instructions of operations on reals and integers are different, specific rules are needed to convert the type of operands by the compiler. Such a conversion of type is called *coercion* [2].

If type coercion is required, the type checker in the μModelica compiler will insert a conversion operator in the expression. For example, the type checking rule for the + (plus) operator is defined as follows:

```
typeOf(E1 + E2):
  if E1.type=IntegerType and E2.type=IntegerType:
    return IntegerType
  elif E1.type=RealType and E2.type=RealType:
    return RealType
  elif E1.type=StringType and E2.type=StringType:
    return StringType
  elif E1.type=RealType and E2.type=IntegerType:
    replace E2 by RealOf(E2)
    return RealType
  elif E1.type=IntegerType and E2.type=RealType:
    replace E1 by RealOf(E1)
    return RealType
  else:
    return TypeError
```

3.7.3 Specification of Type Checking in the μModelica Compiler

In the μModelica compiler, type checking is performed on the equation part of the flat Modelica model. The type checker verifies that each equation and all expressions in that equation tree are type compatible, and will perform type coercion if applicable. The following specifications have been implemented:

```
1. Normal equation (RegularEquation): LHS = RHS
  if LHS.type==RHS.type:
    return Void
  elif LHS.type==IntegerType and RHS.type==RealType:
    replace LHS by RealOf(LHS)
    return Void
  elif LHS.type==RealType and RHS.type==IntegerType:
    replace RHS by RealOf(RHS)
    return Void
  else
    return TypeError

2. Identifier (IdentExp): E
  return E.type

3. Integer (IntExp):
  return IntegerType

4. Real number (RealExp):
  return RealType

5. String (StringExp):
  return StringType

6. Boolean (BoolExp):
  return BooleanType

7. Logical exp: E1 op E2, where op: and (AndExp), or(OrExp)
  if E1.type==BooleanType and E2.type==BooleanType:
    return BooleanType
  else:
    return TypeError

8. Logical exp: not E (NotExp)
  if E.type==BooleanType:
    return BooleanType
  else:
    return TypeError

9. Relation exp: E1 op E2, where op: <, <=, >, >=, ==, <>
  if E1.type==RealType and E2.type==RealType:
    return BooleanType
  elif E1.type==IntegerType and E2.type==IntegerType:
    return BooleanType
  elif E1.type==StringType and E2.type==StringType:
    return BooleanType
  elif E1.type==BooleanType and E2.type=BooleanType:
    return BooleanType
```

```
  elif E1.type=RealType and E2.type=IntegerType:
    return RealType
  elif E1.type=IntegerType and E2.type=RealType:
    return RealType
  else:
    return TypeError

10. Sum exp (SumExp): E1 + E2
  if E1.type=IntegerType and E2.type=IntegerType:
    return IntegerType
  elif E1.type=RealType and E2.type=RealType:
    return RealType
  elif E1.type=StringType and E2.type=StringType:
    return StringType
  elif E1.type=RealType and E2.type=IntegerType:
    replace E2 by RealOf(E2)
    return RealType
  elif E1.type=IntegerType and E2.type=RealType:
    replace E1 by RealOf(E1)
    return RealType
  else:
    return TypeError

11. Exp: E1 op E2, where op: -, *, /
  if E1.type=IntegerType and E2.type=IntegerType:
    return IntegerType
  elif E1.type=RealType and E2.type=RealType:
    return RealType
  elif E1.type=RealType and E2.type=IntegerType:
    replace E2 by RealOf(E2)
    return RealType
  elif E1.type=IntegerType and E2.type=RealType:
    replace E1 by RealOf(E1)
    return RealType
  else:
    return TypeError

12. Unary minus exp: -E
  if E.type=IntegerType:
    return IntegerType
  elif E.type=RealType:
    return RealType
  else:
    return TypeError

13. Function call: E1(E2)
  if E2.type matches the type of declared input variables:
    return E1.type (types of declared output variables)
```

```
    else:
      return TypeError
```

Type checking of some Modelica constructs is not implemented in the current μModelica compiler. The above type checking rules works for continuous models.

4 The Back End

A model described in Modelica is translated to a model in flat Modelica by the Front End. Basically, a flat Modelica model can be seen as a set of DAEs. The ultimate purpose of simulation is to solve such a set of equations.

One can solve a set of DAEs using a DAE solver, e.g., DASSL [22]. But DAE solvers are inefficient. A far more efficient approach is to perform DAE transformations. The purpose of these transformations is to obtain a causal representation of the equations, which might include ODEs and algebraic equations. This set of causal equations can be solved more efficiently with ODE solvers.

This chapter discusses the formula manipulation techniques implemented in the μModelica compiler, which includes *canonical transformation*, *causality assignment*, *equation sorting*, and *detecting algebraic loops*.

4.1 Canonical Transformation

Usually an equations is represented as a tree made up of operators and operands. The *canonical representation* of an equation means that the equation is stored internally in a particular, *unique* way. More specifically, the equation is rewritten in such a way that

- Constants are folded;
- Operators and operands at every level of the tree are in a unique order;
- A few other simplification rules are implemented, the details of which are given below.

4.1.1 Why Canonical Representation?

Even though canonical transformation reuses compile-time efficiency, it is necessary to perform such a transformation for the following reasons:

- For simulation run-time efficiency. If constants are folded at compile-time, there is no need to calculate the same operations on these constants at each time step at simulation-time. For example, assume that a model contains the following equation

  $$a = 2 + 3 + b$$

  If $2+3$ is computed at compile time, e.g., $a = 5 + b$, the same operation need not be computed at each time step during simulation run-time. Therefore, it is a tradeoff between compile-time and run-time. Because the number of simulation runs is far greater than the number of times a model is compiled, it is worth to do so.

- The need for causality assignment. For example, the following equation

$$x+x+y=0$$

cannot be transformed to causal form correctly if unknown `x` is to be calculated based on the value of `y`, e.g., `x` on the left hand side of the equation. In such a case, like terms have to be combined, e.g, $x+x$ is converted to $2x$.

4.1.2 Defining the Canonical Order

An ordering relation on a set of operators is defined so that the nodes in an equation tree can be sorted into the canonical order. The set of operators currently includes sum (+), multiplication (*), power (^), and function calls. The ordering of these operators is defined as follows:

```
'+' < '*' < '^' < 'f()'
```

where `f()` represents function calls. The ordering relation between different function calls is determined by lexicographic ordering of the names of function calls. For example, natural logarithm $log()$ has a higher order than e-based exponent $exp()$. That is,

```
'exp()' < 'log()'
```

The *canonical representation* of an expression or an equation is obtained by sorting the children of every node in the equation tree, together with constant folding and some simplification rules.

4.1.3 Simplification Rules

The paper [26] suggests a set of simplification rules for canonical transformation. A subset of these rules have been implemented in μModelica. These rules are specified as follows:

1. The RHS of an equation is moved to the LHS, and the RHS is set to 0.0, eg., $a=b$ is transformed to $a-b=0.0$.
2. All constants are rewritten as real numbers. Fractions are evaluated. For example, 1/2 is simplified to 0.5, x^2 is rewritten as $x^{2.0}$, and $x^{1/2}$ is written as $x^{0.5}$ etc.
3. A negative number or term is rewritten as:
   - $-c \longrightarrow +(-c)$, where c is constant
   - $-E \longrightarrow +(-1.0)*E$, where E is a term
4. Expressions in reciprocal form (divisions) are rewritten in terms of negative powers. For example:
   - $1/y \longrightarrow 1.0*y^{-1.0}$
   - $x/y \longrightarrow x*y^{-1.0}$
   - $z^3/(x^2+2*x*y) \longrightarrow z^{3.0}*(2.0*x*y+x^{2.0})^{-1.0}$
5. Binary operators + and * are converted to n-ary operators. It is feasible to do so because both + and * are commutative and associative. For example:
   - $a+b+c$ can be rewritten as $+(a,b,c)$
   - $a*b*c$ can be rewritten as $*(a,b,c)$

6. Constant folding. All sum, product, power or other known operations of constants are immediately evaluated. Also, the following rules should be applied to remove superfluous zeros and ones:
   - $0.0 + E \longrightarrow E$;
   - $0.0 * E \longrightarrow 0.0$;
   - $0.0^c \longrightarrow 0.0$;
   - $1.0 * E \longrightarrow E$;
   - $E^{1.0} \longrightarrow E$;
   - $E^{0.0} \longrightarrow 1.0$;
   - $1.0^E \longrightarrow 1.0$;
7. Like terms in a sum are collected and their constant coefficients are added (* distributes over +), eg.:
   $$a * x^p + b * x^p \longrightarrow (a+b) * x^p$$
   where *a,b* are constants (or parameters), *x, p* can be constants, variables or expressions.
8. Product of power of the same base is simplified using this rule:
   $$x^p * x^q \longrightarrow x^{p+q}$$
   where *x,q,p* can be constants, variables or expressions.
9. The power of a power can be simplified as:
   $$(x^p)^q \longrightarrow x^{p+q}$$
   where *x,q,p* can be constants, variables or expressions. A further simplification occurs if *x,q* are both constants:
   $$(x^p)^q \longrightarrow (x^q)^p$$
10. The power of a product:
    $$(x * y)^p \longrightarrow x^p * y^p$$
    x,y,p can be constants, variables or expressions, but with *x,y* not being both constants. The opposite of this rule should be applied when both *x* and *y* are constants, and so they can be folded:
    $$x^p * y^p \longrightarrow (x * y)^p.$$
11. A constant multiplying an expression which is a sum of terms containing variables, is distributed:
    $$c * (t_1 + t_2 + \ldots + t_n) \longrightarrow c * t_1 + c * t_2 + \ldots + c * t_n$$
    where *c* is a constant, and t_i is a set of terms.

4.1.4 The Transformation Algorithm

The transformation consists of a series of applications of the simplification rules. These rules are invoked in the following order:

1. According to rule 3 and 4, convert division expressions, and subtraction expressions into multiplications, and sum expressions, respectively.

2. According to rule 5, on both sides of the equation, convert the binary operators + and * into n-ary operators.
3. Move the RHS of the equation to the LHS, and set the RHS to 0.0
4. Apply rule 5 to the resulting LHS again.
5. Constant folding according to rule 6.

After these preliminary transformation steps, the following rules are iterated over until all nodes in the tree are in canonical order:
While the tree changes:

1. Simplify the powers of products by applying rule 10.
2. Apply rule 5 to the LHS to flatten the + and * operators.
3. Constant folding according to rule 6.
4. Sort nodes into canonical order.
5. Apply rule 9 to flatten the powers of powers.
6. Apply rule 8 to simplify the products of powers of the same base.
7. Constant folding according to rule 6.
8. Sort nodes into canonical order.

The iteration stops if there is no further change occurs in the tree. Finally the following rules are applied:

1. Rule 11: distribute constants.
2. Flatten the + and * operators on the LHS.
3. Constant fold.
4. Sort nodes into canonical order.
5. Apply rule 7 to collect like terms.
6. Constant fold again.
7. Sort nodes into canonical order.

4.1.5 An Example

Here is an example showing the canonical transformation in the μModelica compiler. Given the following model as input:

```
class Canonical
  Real a, b, c, d, x;
  Real e, f, g, h;
equation
  2-1-3-4-5=h;
  a+a+2.0*a+b+(c+d*a/b)=a-2*b*c-d;
  2*a^3+3*c^d=a^3-2*c^d;
  a*a*b*c*c*d+b*a^2*a+e^a*f^b*e^(c+d)=0.0;
  e^a*f^b*e^(c+d+g)=0.0;
  ((4/2*a^(0+b))^2)^(2*(3+2))=0;
  2*a^a+a^a+a^2=0;
```

```
  (a*(b+c)^2*d^e*(2*3+4)^2)^3=e;
  (x+b)^2-(c*b)^2+a^2*d=(c+d)^2;
end Canonical;
```

The μModelica compiler rewrites the equation part into the following canonical form:

```
(-11.0)+h*(-1.0)=0.0;
a*3.0+a*b^(-1.0)*d+b+b*c*2.0+c+d=0.0;
a^3.0+c^d*5.0=0.0;
a^2.0*b*c^2.0*d+a^3.0*b+e^(a+c+d)*f^b=0.0;
e^(a+c+d+g)*f^b=0.0;
a^(b*20.0)*1048576.0=0.0;
a^2.0+a^a*3.0=0.0;
a^3.0*(b+c)^6.0*d^(e*3.0)*1000000.0+e*(-1.0)=0.0;
a^2.0*d+(b+x)^2.0+b^2.0*c^2.0*(-1.0)+(c+d)^2.0*(-1.0)=0.0;
```

4.2 Causality Assignment

The real essence of Modelica is non-causal modeling, which is characterized by a set of implicit equations. To solve the various unknowns in the system more efficiently, it is far more preferrable to have a causal representation of equations. It is possible in many cases to transform a non-causal representation into a causal one. Such a transformation is called *causality assignment*. For example, consider the following set of implicit equations:

$$\begin{cases} x+y+z &= 0 \quad \text{Eq 1} \\ x+3z+u^2 &= 0 \quad \text{Eq 2} \\ z-u-16 &= 0 \quad \text{Eq 3} \\ u-5 &= 0 \quad \text{Eq 4.} \end{cases}$$

To compute this set of equations on a computer, each equation must be identified that it is used to solve for what variable. That is, a *matching* between equations and variables must be found. This problem can be solved in terms of graph algorithms. More specifically, it can be solved elegantly by turning it into the problem of finding a *maximum network flow* in a bipartite graph. Equations and variables are turned into nodes and the dependencies between equations and variables are turned into edges. Adding a source node and a sink node to the bipartite graph results in a directed graph. For example, figure 4.1 shows the resulting graph of this set of equations. Causality assignment can be obtained by maximizing the flow from the source node to the sink node. As shown in figure 4.1, the flow paths indicate the correspondence between each variable and the equation used to compute it:

$$\begin{cases} \underline{y} &= -x-z \\ \underline{x} &= -3z-u^2 \\ \underline{z} &= u+16 \\ \underline{u} &= 5 \end{cases}$$

Therefore, the problem to be solved here is to find a maximum flow in a directed bipartite graph. In history, many attempts have been made to solved this problem. Dinic's algorithm in finding such a maximum flow is efficient if all edges in the graph has unit capacity. It is implemented in the μModelica compiler. This section discusses Dinic's algorithm in detail [9].

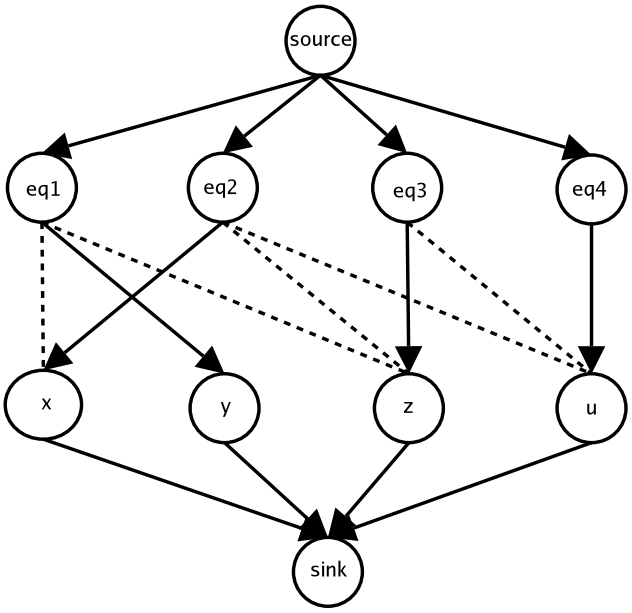

Figure 4.1: Causality Assignment: Network Flow in Bipartite Graph

4.2.1 Flows, Augmenting Paths, and Residual Graph

Before discussing Dinic's algorithm, some important concepts in the theory of network flows are introduced.

Let $G = [V, E]$ be a directed graph made up of the set of vertices $V \equiv \{v\}$ and the set of edges $E \equiv \{e\}$. Two special vertices, the source and the sink, are identified as s and t, respectively. The number of vertices in G is n and the number of edges is m. Every edge is associated with a positive *capacity* $cap(v,w)$. A *flow* f on G is defined as a real-value function on vertex pairs. It has the following properties:

- *Skew symmetry*: $f(v,w) = -f(w,v)$.
  Also, if $f(v,w) > 0$, then there is a flow from v to w.
- *Capacity constraint*: $f(v,w) \leq cap(v,w)$.
  A flow is said to *saturate* the edge $[v,w]$ if the equality $f(v,w) = cap(v,w)$ holds.
- *Flow conservation*: for every vertex v excluding the source s and sink t, the net incoming flow must equal the net outgoing flow: $\sum_{w \in V} f(v,w) = 0$.

The *residual capacity* for a flow f in a network is also given by a function on vertex pairs. It is the difference in the capacity of the edge connecting the two vertices and the flow across the edge:

$$res(v,w) = cap(v,w) - f(v,w). \tag{4.1}$$

An amount of $res(v,w)$ additional units of flow can be pushed from v to w by increasing the flow $f(v,w)$ and correspondingly decreasing $f(w,v)$. We can construct the *residual graph* R for a flow f, which is the graph with vertex set V including the source s and sink t, and an edge $[v,w]$ of capacity $res(v,w)$, such that this capacity is positive: $res(v,w) > 0$.

An *augmenting path* for f is defined as a path p from s to t in R. The *residual capacity* of this path, denoted by $res(p)$, is the minimum value of $res(v,w)$ for $[v,w]$ an edge of p. The value of the flow f can be increased by any amount Δ up to $res(p)$ by increasing the flow on every edge of p

by Δ. To satisfy the property of symmetry, if a change of Δ is made to $f(v,w)$, there should be a corresponding amount of $-\Delta$ made to $f(w,v)$.

The concepts of *blocking flow* and *level graph* are important to understand Dinic's algorithm. A flow f is a *blocking flow* if every path from the source s to the sink t contains a saturated edge (an edge $[v,w]$ is said to be *saturated* if $f(v,w) = cap(v,w)$). There is no way to increase the value of a *blocking flow* by increasing additional flow along any path in the graph. However, it is possible to do so by rerouting, which means the flow on some edges is decreased while it is increased in other edges. Let R be the residual graph of a flow f. The *level* of a vertex v in R is the length of the shortest path from the source node s to v. The *level graph* L for f is the subgraph of R containing only the vertices reachable from s, and only the edges $[v,w]$ such that $level(w) = level(v) + 1$. L contains every shortest augmenting path and can be constructed in $O(m)$ time by *breadth-first search*.

4.2.2 Dinic's Algorithm

Dinic's algorithm is implemented in the μModelica compiler to solve the problem of causality assignment. It is to find a *maximum flow* from the source node to the sink node on a directed graph. Given a directed graph with a source and a sink, it starts with zero flow and repeats the *blocking step* until the sink t is no longer in the level graph for the current flow.

The *blocking step* in Dinic's algorithm is defined as follows:

- Find a blocking flow f'on the level graph for the current flow f.
- Replace f by the flow $f + f'$ defined by: $(f + f')(v,w) = f(v,w) + f'(v,w)$.

The remaining problem is to find a blocking flow. We also adopt Dinic's method: let G be the input acyclic graph, use depth-first search (DFS) to find a path from the source node s to the sink node t, push along the path the amount of flow that saturate the edge with smallest residual, then delete all newly saturated edges, and repeat this procedure until t is not reachable from s. The algorithm is described more formally below:

- **Initialize**: Let $p = [s]$ and $v = s$. Go to **Advance**.
- **Advance**: If there is no outgoing edge from v, go to **Retreat**. Otherwise, let $[v,w]$ be an outgoing edge of v. Replace p by $p + [w]$, and v by w. If $w \neq t$ repeat **Advance**; if $w = t$, go to **Augment**.
- **Augment**: Let δ be $min(cap(v,w) - f(v,w))$ where $[v,w]$ is any edge in path p. Add δ to the flow of very edge on p, delete from G all newly saturated edges, and go to **Initialize**.
- **Retreat**: If $v = s$, halt. Otherwise, let $[u,v]$ be the last edge on p. Delete v from p and $[u,v]$ from G, replace v by u, and go to **Advance**.

It can be proved that Dinic's algorithm above correctly finds a blocking flow in $O(nm)$ time, and a maximum flow in $O(n^2 m)$ time. It can also be proved that on a unit network, Dinic's algorithm finds a blocking flow in $O(m)$ time, and a maximum flow in $O(n^{1/2} m)$ time. In a unit network, all edge capacities are integers, and each vertex v other than the source and the sink has either a single entering edge of capacity one, or a single outgoing edge of capacity one.On a network whose edge capacities are all one, Dinic's algorithm finds a maximum flow in $O(\min\{n^{2/3} m, m^{3/2}\})$ time [13].

4.2.3 ODEs in Causality Assignment

In Modelica, the *time derivative* of a state variable is introduced by the operator $der()$. An *ordinary differential equation* (ODE) has the following form (in Modelica syntax):

$$der(x) = f(x)$$

The numerical approximation of the time derivative of a variable is defined as:

$$der(x) = (x - x_{old})/\Delta t$$

The value of x can be computed by numerical integration methods.

Because either the value of x can be computed via integration based on $der(x)$, or the value or $der(x)$ can be derived from the current value of x and x_{old}, only one of x or $der(x)$ is treated as unknown in causality assignment. How the μModelica compiler handles ODEs in causality assignment is discussed in this section. More specifically, an algorithm to choose which form of a state variable as unknown is given.

Integral Causality and Derivative Causality

In causality assignment, *integral causality* means that the time derivative of a state variable is chosen as unknown, while the state variable itself is computed through numerical integration. *Differential causality* works in the other way around. That is, a state variable itself is chosen as known, and the time derivative of the state variable is computed through numerical differentiation.

In fact, *integral causality* is more preferrable in simulation computation since it gives more stable simulation results. But in some cases, choosing integral causality might lead to failure in causality assignment. Consider the following example:

$$\begin{cases} y &= sin(time) \\ der(x) &= y+z \\ der(y) &= x+z \end{cases} .$$

If $der(x)$, $der(y)$, and z are chosen as unknowns, a valid causality assignment result can not be found because both sides of the first equation are known. However, it is possible to find a valid causality if some of the unknowns with integral causality are replaced by differential causality. For example, if state variable y is assigned derivative causality, a valid causality assignment can be found.

The Algorithm

An algorithm for handling ODEs in causality assignment is implemented in the μModelica compiler. Based on the fact that integral causality can give more accurate simulation result, the algorithm prefers integral causality as many as possible. The algorithm is described more formally as follows:

- By default, integral causality is chosen for all state variables. For example, $der(x)$, $der(y)$, and z are regarded as unknowns by default in the previous system.
- If a valid causality assignment is found, return.
- Otherwise, a list of all possible combinations of integral causality and derivative causality is generated. In the previous example, all possible combinations are:
  - $der(x), y, z$

- $x, der(y), z$
- x, y, z

- Begin with the combination with least derivative causality. If causality assignment still fails, try the next combination in the list that has the least derivative causality. Repeat this step until a valid causality is found.
- If finally causality assignment fails after all the combinations have been tried, a DAE solver is called to solve the set of implicit equations directly.

This algorithm is a heuristic approach to finding the most appropriate combination of integral causality and derivative causality. In the worst case, the algorithm's complexity is combinatorial. We realize that there might exist a more direct and efficient approach. But ours is an easy-to-understand approach. At the time being for the sake of fast prototyping, we chose this approach because it is easy to implement.

Example

Consider the following Modelica model encoding the equations given above:

```
class ODE1
  Real x, y, z;
equation
  y=sin(time);
  der(y)=x+z;
  der(x)=y+z;
end ODE1;
```

The μModelica compiler generates the following causality assignment result for this model:

```
-------causality assignment result -------
Variables: der(y), der(x), z,
der(x)+y*(-1.0)+z*(-1.0) = 0.0    is used to solve for 'der(x)'
der(y)+x*(-1.0)+z*(-1.0) = 0.0    is used to solve for 'der(y)'
***Invalid causality!***

-------causality assignment result -------
Variables: y, der(x), z,
sin(time)*(-1.0)+y = 0.0    is used to solve for 'y'
der(x)+y*(-1.0)+z*(-1.0) = 0.0    is used to solve for 'der(x)'
der(y)+x*(-1.0)+z*(-1.0) = 0.0    is used to solve for 'z'
***Valid causality!***
```

From this output, one can see that the compiler first tried with all integral causality but failed. Then it succeeded in finding a valid causality assignment result for the model when variable y is given derivative causality.

Inserting Derivative Equation and Integration Equation

The μModelica compiler, inserts a derivative equation of the following form:

$$der(x) = (x - x_{old})/\Delta t$$

for every variable that was given derivative causality. An integration equation of the following form:

$$x = integration(x_{old}, der(x))$$

is inserted for every variable that was given integral causality. For example, inserting these types of equations leads to the following complete computation model for the previous system:

$$\begin{cases} y &= sin(time) \\ der(y) &= (y - y_{old})/\Delta t \\ z &= x_{old} - der(y) \\ der(x) &= y + z \\ x &= integration(x_{old}, der(x)) \end{cases} .$$

These additional equations are inserted at code generation time after the equations are sorted. That is, they are not taken into account in sorting.

4.3 Sorting of Equations

Even though the original set of DAEs has been transformed to a causal representation, in general they are not yet in a correct computation order. The following example (set of equations) illustrates this problem when a mathematical sets of equations are coded in a programming language with sequence semantics such as C, where $sin(time)$ is considered as known:

$$\begin{cases} a = & b^2 + 3 \\ b = & sin(c * e) \\ c = & (d - 0.5)^{0.5} \\ d = & 1/2 \\ e = & sin(time) \end{cases}$$

If it is coded in the above sequence, uninitialized variables will be given a zero value which leads to erroneous results:

$$\begin{cases} a = & 3 \\ b = & 0 \\ c = & -0.5^{0.5} \text{(exception)} \\ d = & 1/2 \\ e = & sin(time) \end{cases}$$

However, it is possible to compute the correct solution of the set of equations if they are re-arranged in the following sequence:

$$\begin{cases} d = & 1/2 \\ e = & sin(time) \\ c = & (d - 0.5)^{0.5} \\ b = & sin(c * e) \\ a = & b^2 + 3 \end{cases}$$

Therefore, the equations must be sorted in the reverse order of their dependencies, i.e., if to compute the value of an unknown it is necessary to know the value of another variable, then the latter variable must be computed prior to this one. This section presents the algorithm for sorting the equations into a correct computation order.

4.3.1 Dependency Graph

Before equations are sorted, the computation dependency graph is built. Each vertex in the graph represents a variable to be computed. An edge from vertex *a* to vertex *b* means that the value of *a* depends on the value of *b*, i.e., *b* appears on the RHS of the equation to compute *a*. For example, the computation dependency graph of the set of equations given above is shown in figure 4.2.

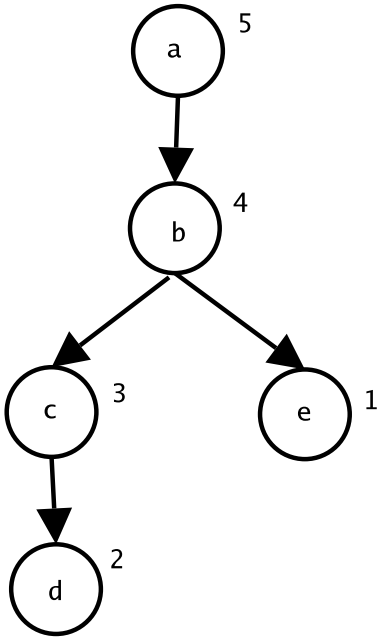

Figure 4.2: Sorting of Equations: Dependency Graph

4.3.2 The Algorithm

Based on the graph of computation dependency, the sorting of equations can be achieved by a *topological sort* with post-order numbering on this graph. The numbers indicate the order in which equations are computed. In the μModelica compiler, the following algorithm is implemented to determine the order in which equations need to be written:

```
# topSort() and dfsLabelling() both refer to the global counter,
# dfsCounter, which will be incremented during the topological sort.
# It is used to assign an orderNumber to each node in the graph.
dfsCounter = 1

# topSort() performs a topological sort on a directed graph
# (either acyclic or cyclic)
def topSort(graph G)
  # mark all nodes as unvisited
  for nods in G
    node.visited = false
  # start dfsLabelling() from any node in the graph until all
  # nodes have been visited
  for node in G
```

```
    if node.visited == false
      dfsLabelling(node)

# dfsLabelling() performs a depth-First traversal of a possibly
# cyclic directed graph. Nodes are labelled with numbers.
def dfsLabelling(node n, graph G)
  if node.visited == false
    # mark the node as visited
    node.visited = true
    # perform dfsLabelling() on all neighbours
    for neighbour in node.out_neighbour
      dfsLabelling(neighbour, G)
    # label the node with the counter and subsequently increment
    # the counter
    node.orderNumber = dfsCounmter
    dfsCounter ++

# The program terminates when all nodes have been visited.
# As a result, all nodes are labelled with numbers which
# indicate the order of computation.
```

Figure 4.2 also shows the result of sorting. The numbers beside vertices indicate the order of computation. The result of sorting is not unique. Figure 4.3 gives another correct computation order.

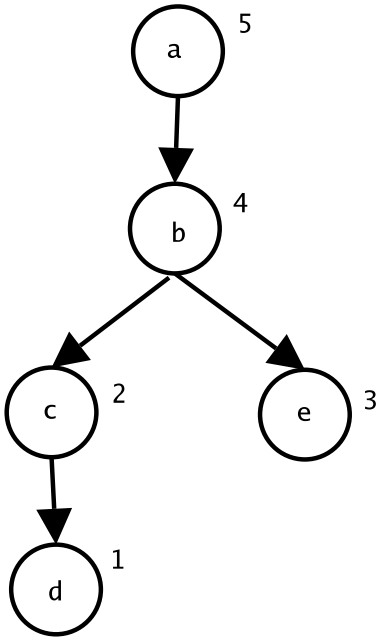

Figure 4.3: Sorting of Equations: Another Sorting Result

4.4 Algebraic Loop Detection

In some cases, sorting is not possible due to the existence of dependency cycles (algebraic loops). For example, the following set of equations

$$\begin{cases} x = & y + 16 \\ y = & -x - z \\ z = & 5 \end{cases}$$

can not be sorted since there exists a dependency cycle between x and y, or in other words, the equations to calculate x and y form an algebraic loop. Therefore, before sorting equations, detecting algebraic loops is required. Once detected, the equations involved should be isolated, and be solved simultaneously either with symbolic or numerical methods.

4.4.1 The Algorithm

Detecting algebraic loops (finding dependency cycles) can be turned into the problem of locating *strongly connected components* in a graph. A strongly connected component is a set of nodes in a graph whereby each node is reachable from each other node in the strongly connected component. Based on the result of the previous topological sort with post-order numbering, this problem can be solved by producing a list of strongly connected components. If a node is not in a cycle, it will be a strongly connected component with only itself as a member. Therefore, if there exist algebraic loops, some of the strongly connected components in the produced list must contain more than one node. The algorithm for locating strongly connected components is given below.

```
# Producing a list of strongly connected components.
# Strongly connected components are given as lists of nodes.
def strongCom(graph G)
  # Perform a topological sort in the graph with post-order
  # numbering (the algorithm is given in the previous section)
  topSort(G)
  # Produce a new graph with all edges reversed.
  rev_graph = reverse_edges(G)
  # Start with an empty list of strong components
  strong_components = []
  # Mark all nodes as not visited
  for node in rev_graph:
    node.visited = false
  # As strong components are discovered and added to the
  # strong_components list, they will be removed from rev_graph.
  # The algorithm terminates when rev_graph is reduced to empty.
  while rev_graph != empty:
    # Start from the highest numbered node in rev_graph
    start_node = highest_orderNumber(rev_graph)
    # Perform a depth first search on rev_graph starting from
    # start_node, collecting all nodes visited.
    # This collection (a list) will be a strong component.
    # dfsCollect() also marks nodes as visited to avoid infinite
    # loops.
```

```
component = dfsCollect(start_node, rev_graph)
# Add the found strong component to the list of strong
# components.
strong_components.append(component)
# Remove the identified strong component
rev_graph.remove(component)
```

If a subset of equations are located in the same strongly connected component, they will be identified as an *algebraic loop*. This subset of equations need not to be rewritten into causal form. Instead, they will be solved simultaneously with numerical methods.

Causality assignnment, sorting, and algebraic-loop detection can also be carried out by transforming the DAEs into the *block-lower-triangular* (BLT) form. Algebraic loops can be more easily identified in the BLT form. But using a dependency graph to detect algebraic loops is a clean and didactic way to illustrate the problem and it enables visualization (for small problems).

In most cases, a Modelica model is finally transformed to a set of causally represented equations with correct computation order. Such a set of equations may contain algebraic loops. Integrators and algebraic loop solvers are required to compute the solution of the equations. In the worst case, if formula manipulation fails, a DAE solver is required. But it is a far less efficient approach than the previous one.

4.5 Design and Implementation

There are different approaches to implementing causality assignment, sorting, and algebraic-loop detection. The *Block-lower-triangular* (BLT) transformation approach has been implemented in some tools such as the PELAB openModelica compiler. It is a technique based on matrix transformation. Our implementation of these problems is purely based on graph algorithms. This section presents the data structure and some issues in our implementation of the Back End.

4.5.1 The Data Structure

As mentioned earlier during the description of Dinic's algorithm, the causality assignment problem is turned into the problem of finding a maximum flow in a bipartite graph. Such a bipartite graph consists of different types of nodes, and edges, as depicted in Figure 4.4.

A class hierarchy of nodes is defined. There are 3 types of nodes in a bipartite graph: the ones that represent equations, the ones that represent variables, and the source and sink. The source and the sink are instances of the parents class `Node`. Equations and variables are represented by `EqNode` and `VarNode`, respectively.

A bipartite graph is an instance of `FlowGraph`, which consists of a source, a sink, a list of `EqNode`, and a list of `VarNode`. Also, there is a class called `FlowEdge` representing outgoing edges from each node. It is similar to the representation of an adjacency list. That is, each node has a list of outgoing edges. Each edge specifies the destination node, as well as edge capacity and currently flow.

Remember that in Dinic's algorithm, a *level graph* is created based on the residual graph. It is a subgraph of the residual graph. It contains a subset of nodes of the residual graph, but the nodes are connected by different edges. In our implementation, an object instance of `FlowGraph` is used to represent both flow/residual graph and level graph. The way that these two graphs are distinguished is that, each node keeps two different lists to store its outgoing edges in a flow/residual graph and its outgoing edges in a level graph, respectively. Even though this approach might cause more

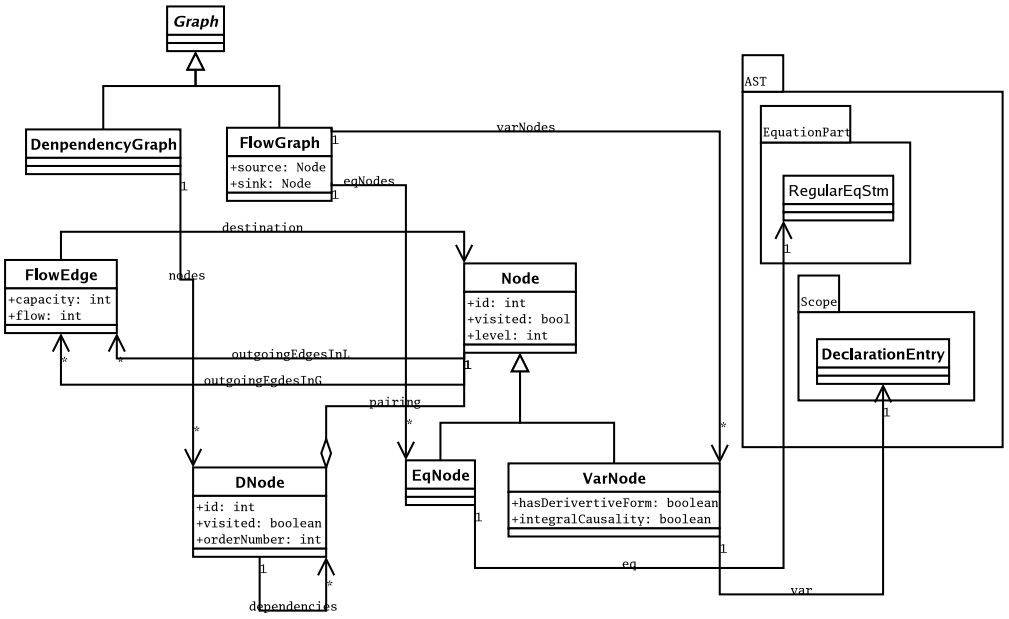

Figure 4.4: The Data Structure for Causality Assignment

coherence between the residual graph and the level graph, it is more efficient, in the rerouting step, to combine a new blocking flow into the existing one in the residual graph.

Sorting is executed based on computational dependency, which is represented by a dependency graph. Causality assignment gives one-to-one pairings (via saturated edges) between equation nodes (`EqNode`) and variable nodes (`VarNode`) in the bipartite graph. Unsaturated edges from equation nodes to variable nodes indicate computational dependency. A dependency graph is constructed by turning each pairing (consists of an `EqNode` and an `VarNode` linked by an saturated edge) into a `DNode`, and unsaturated edges into dependency edges. The design of these classes is shown in Figure 4.4.

4.5.2 Implementation Issues

Finding a blocking flow is the key issue in solving the problem of causality assignment in terms of Dinic's algorithm. This section presents our implementation (pseudocode) of the *blocking step* in Dinic's algorithm.

Level Graph Construction

We are to find a blocking flow in a level graph. Before discussing the implementation of how to find a blocking flow, we first show how a level graph is constructed.

In a residual graph R, the *level* of a vertex v is the length of the shortest path from the source node s to v. The *level graph* L for a flow f is the subgraph of R containing only the vertices reachable from s, and only the edges $[v,w]$ such that $level(w) = level(v) + 1$. L can be constructed in $O(m)$ time by *breadth-first search*. The pseudocode of constructing a level graph is given as follows:

```
# G::FlowGraph
# G represents the residual graph
def buildLevelGraph(G):
    # initialization
```

```
    for node in G.getAllNodes():
        node.deleteAllOutgoingEdgesInL()
        node.setVisited(false)
        node.setLevel(0)
    G.isSinkInL=False
    G.source.setVisited(True)
    L0=[G.source]
    # a global counter of the current level during graph traversal
    level=0
    G.source.setLevel(level)
    # graph traversal by breadth-first search
    while L0 not empty:
        level=level+1
        # another list for breadth-first search
        L1=[]
        for v in L0:
            for each unsaturated outgoing edge [v, w]:
                # add edge [v,w] to L if w is not visited
                if not w.isVisited():
                    w.setVisited(True)
                    w.setLevel(level)
                    # set the flag if L contains the sink node
                    if w==G.sink:
                        G.isSinkInL=True
                    L1.append(w)
                    v.addOutgoingEdgeInL([v,w])
                # add edge [v,w] to L if level(w)=level(v)+1
                else:
                    if v.level+1==w.level:
                        v.addOutgoingEdgeInL([v,w])
        L0=L1
    # now G also contains the information of the level graph
    return G
```

A Blocking Step

A blocking step in Dinic's algorithm consists of finding a blocking flow f' on the level graph L with the current flow f, and replace the current flow f by $f+f'$. The following method implements a blocking step. The input to the method is a residual/level graph with flow f, and the output is the same graph with flow $f+f'$.

```
# Find a blocking flow f' on the level graph L for the
# current flow f,
# and replace the current flow by f+f'
def blockingStep(L):
    # initialize
    p=[L.source]
    v=L.source
    while True:
```

```
        # Advance
        outgoingEdges=v.getAllUnsaturatedEdgesInL()
        # if v has outgoing edges
        if outgoingEdges is not empty:
            # by default, pick the first edge [v,w] in the list
            [v,w]=outgoingEdges[0]
            # replace path p by p+w, and v by w
            p.append(w)
            v = w
            if v is not L.sink:
                # repeat Advance
                continue
        # Retreat
        else:
            if v is L.source:
                # halt
                return L
            else:
                [u,v]=last edge of p
                delete [u,v] in L
                delete v in p
                # replace v by u
                v = u
                continue
        # Augment
        # compute the saturated flow delta on p
        delta=saturatedFlow(p)
        for each edge (e) on p:
            # add delta to the flow of every edge on p
            e.addFlow(delta)
            # add -delta to the flow of corresponding reversed edge
            e1=reversed(e)
            e1.addFlow(-delta)
            #delete newly saturated edge in L
            if e.getCap()==e.getFlow():
                delete e in L
        # initialize
        p=[L.source]
        v=L.source
    return L
```

4.5.3 Extension to Hybrid Systems

Even though this thesis does not cover hybrid behavior in Modelica, it is important to consider that the current data structure is capable of supporting hybrid models. That is, the current data structure should be capable of handling all Modelica features from syntax to semantics, and symbolic transformation in the Back End.

It has been mentioned earlier that the μModelica compiler provides full support of Modelica syntax.

The abstract syntax covers all Modelica constructs. The semantics of hybrid models is also defined in terms of the translation of original Modelica source file into DAEs, while some of the DAEs are conditionally evaluated. The current data structure supports conditional equstions. This should enable the translation of hybrid models into flat Modelica.

Causality assignment is more complicated with hybrid behavior. Our implementation of causality assignment is based on a very general graph data structure. It is hard to predict that the current data structure will fully support hybrid systems in causality assignment. But we are confident that using graph algorithms is general enough (as opposed to matrix-based approaches), and it will work with hybrid models, possibly with additional structures.

5 Code Generator

As it has been mentioned, The μModelica compiler project is an open-source project, and it is based entirely on all freely available resources in the public domain. To meet the requirement that one can simulate Modelica models with free resources, a free simulator is also needed. *GNU Octave* is a high-level language and environment which is primarily intended for numerical computation. It is intended as a free alternative to Matlab. It can solve linear and nonlinear problems numerically, and can perform other types of numerical experiments. It is a freely available software. The μModelica compiler currently generates Octave code. This approach is far from optimal, and the flavor of the generated code is more suited for *Simulink S-functions* or *DSblock* [20]. This chapter presents how Octave is used to simulate Modelica models.

5.1 Problems to be Solved

After formula manipulation has been performed in the Back End, a set of implicit DAEs are possibly transformed to a set of explicit (causal) equations, which might contain linear equations, ODEs and integration equations, and algebraic loops (linear or nonlinear). This section discusses how Octave solves each of these problems.

5.1.1 Integrating ODE

Octave is able to solve nonlinear differential equations of the following form:

$$\frac{dx}{dt} = f(x,t)$$

with initial condition

$$x(t_0) = x_0.$$

Users must specify the function $f(x,t)$, i.e., the RHS of the equation, for Octave to integrate the equation. Consider the following example:

```
function xdot = f(x, t)
  a = 1.5
  b = 2.0
  c = 3.0
  d = 6.0
  xdot(1) = a*x(1)+b*x(1)*x(2)^2
  xdot(2) = c^2*x(1)+d*x(2)*x(1)^2
endfunction
```

This function will solve the following set of nonlinear ODEs

$$\begin{cases} \frac{dx_1}{dt} = & ax_1 + bx_1x_2^2 \\ \frac{dx_2}{dt} = & c^2x_1 + dx_2x_1^2 \end{cases}$$

Users also need to specify the time space and the time step over which the differential equations are integrated. For example,

```
t = linspace(0, 50, 200)
```

defines the set of output times as a column of vector, where 0 is the initial time, 50 is the end time of integration. This time space is divided into 200 intervals. Given the initial condition

```
x0 = [1; 2]
```

the set of differential equations can be solved by calling the built-in *lsode* function

```
x = lsode("f", x0, t).
```

The return value is a matrix of size 2×200. The first column of the matrix corresponds to the value of x_1, and the second column corresponds to the value of x_2, at each time step. The output at initial time corresponds to the initial condition given above.

5.1.2 Solving Nonlinear Equations

During formula manipulation in the Back End, algebraic loops are identified and the equations involved are not transformed to causal representation. An algebraic loop can either be linear or nonlinear. Octave can solve sets of nonlinear equations of the form

$$F(x) = 0$$

using the function *fsolve*, which is defined as follow:

$$[x, info, msg] = fsolve(fcn, x_0)$$

where fcn is the name of a function of the form $f(x)$, x_0 is an initial guess value of x. For example, the function to solve the following set of nonlinear equations:

$$\begin{cases} 2x^2 + 3xy + y^2 = & 5 \\ 3x - 2xy^2 + 2y^3 = & 2 \end{cases}$$

is written as :

```
function y = f(x)
  y(1) = 2*x(1)^2+3*x(1)*x(2)+x(2)^2-5
  y(2) = 3*x(1)-2*x(1)*x(2)^2+2*x(2)^3-2
endfunction
```

To solve this set of equations, one must give an initial guess of $x(1)$ and $x(2)$. For example,

```
x0 = [0; 0].
```

Then call *fsolve* to find the roots of the system

```
[x, info] = fsolve("f", [0;0])
```

A return value of $info = 1$ means that the solution has converged.

5.2 The Structure of the Simulation Process

Figure 5.1 depicts the structure of the simulation process.

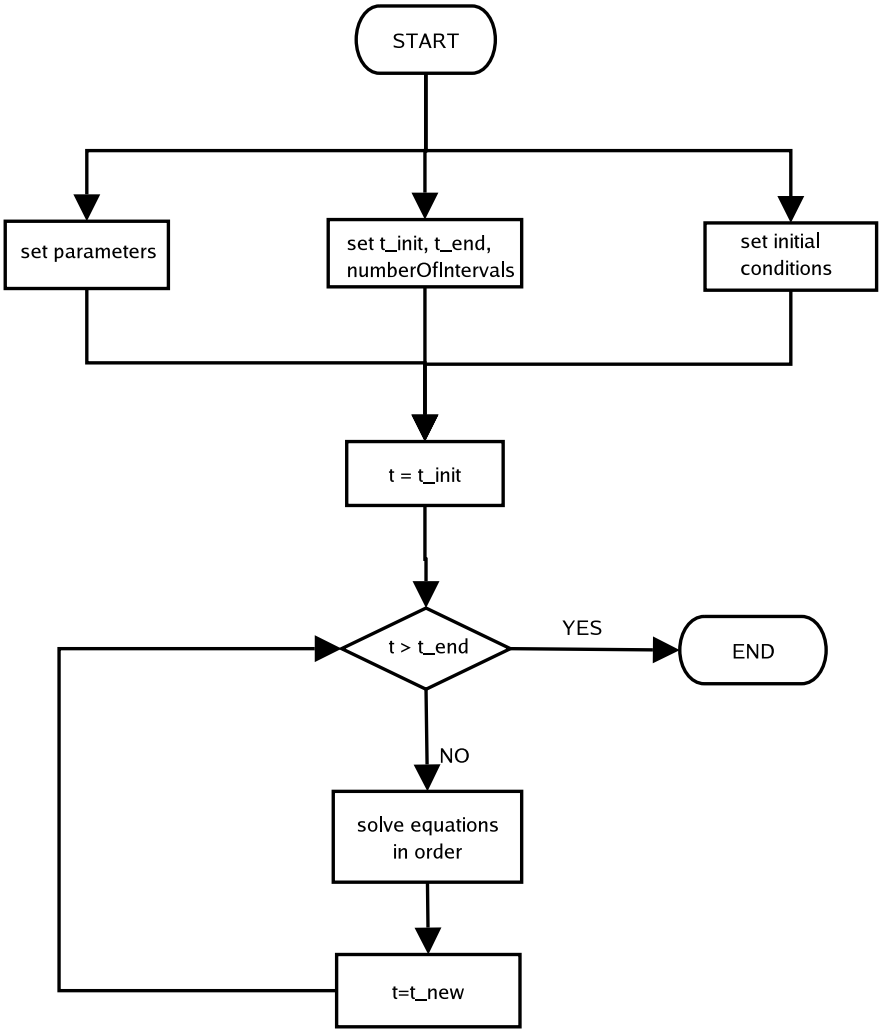

Figure 5.1: Structure of the Simulation Process

Consider the following example:

```
model Equation
  constant Real pi=3.1416;
  parameter Real a=2.0;
  Real w, x, y, z;
equation
  x = sin(time);
  der(y)+x = y^2;
  w+z+x = 3;
  w-2*z = 1;
end Equation;
```

Given this model as input, the μModelica compiler transforms the equations into a causal form:

```
x=a*pi+sin(time)
der(y)=(-1.0)*x+y^2.0
Algebraic Loop 1:  -3.0+w+x+z = 0.0
Algebraic Loop 1:  -1.0+w+z*(-2.0) = 0.0
```

and generates the following Octave code:

```
# set time
time_init=input("Please enter initial time: ");
time_end=input("Please enter end time: ");
time_step=input("Please enter time step: ");
num_of_intervals=(time_end - time_init)/time_step;
time = linspace(time_init, time_end, num_of_intervals)';
# Constants
global pi_last=3.1416
# Parameters
global a_last=2.0
a_last=input("enter parameter value: a(2.0)");
# Variables and model initialization
global z=zeros(num_of_intervals, 1);
z(1)=input("Please enter initial value of z: ");
global z_last=z(1);
global w=zeros(num_of_intervals, 1);
w(1)=input("Please enter initial value of w: ");
global w_last=w(1);
global y=zeros(num_of_intervals, 1);
y(1)=input("Please enter initial value of y: ");
global y_last=y(1);
global der_y=zeros(num_of_intervals, 1);
der_y(1)=input("Please enter initial value of der_y: ");
der_y_last=der_y(1);
global x=zeros(num_of_intervals, 1);
x(1)=input("Please enter initial value of x: ");
global x_last=x(1);

function loop1 = f_loop1(x)
  global x_last;
  loop1(1)=(-3.0)+x(2)+x_last+x(1);
  loop1(2)=(-1.0)+x(2)+x(1)*(-2.0);
endfunction

function y_dot = f_y(yi, ti)
  global x_last;
  global y_last;
  y_dot(1)=(-1.0)*x_last+yi(1)^2.0;
endfunction

for i=2:num_of_intervals
```

```
    # Equation 1
    x(i)=a_last*pi_last+sin(time(i));
    x_last=x(i);
    # Equation 2
    der_y(i)=(-1.0)*x_last+y_last^2.0;
    der_y_last=der_y(i);
    time_i=linspace(time(i-1), time(i), 10)';
    yi0=[y_last];
    y_i=lsode("f_y", yi0, time_i);
    y(i)=y_i(10);
    y_last=y(i);
    # Equation 3
    init_guess=zeros(2, 1);
    [loop1, info]=fsolve("f_loop1", init_guess);
    z(i)=loop1(1);
    z_last=z(i);
    w(i)=loop1(2);
    w_last=w(i);
  endfor
```

From this sample, we can see that the simulation process consists of the following steps:

- Set the initial time, end time, and time steps of a simulation run;
- Set up parameters for a simulation run;
- Set up initial conditions;
- Define functions to compute integration equations and algebraic loops;
- Compute the value of unknowns in order, at each time step.

Note that the main simulation loop advances simulation time in fixed time-steps. Within each time step, Octave solvers, including *lsode* get called. Obviously, this use of Octave is overkill. The structure of the generated code (without the time-step loop) is closer to that of Simulink S-functions.

5.2.1 Time Setup

Before a simulation run, users are prompted to enter the initial time (usually 0), the end time, and the number of intervals that this time space is divided into. Smaller time step leads to more accurate simulation result, at the cost of lower run time efficiency.

5.2.2 Constants, Parameters, and Variables

In simulation models, it is meaningful to specify the *variability* of identifiers. As in some other modeling and simulation languages, three levels of variability are identified in Modelica:

1. Constant: the value of a constant never changes after it is declared. Wherever the identifier occurs, it may be replaced by its value. Here in the μModelica compiler, substituting constant values, which is called *constant propagation* in compiler theory, is not implemented. The value of a constant identifier is evaluated at run time. But this is left as future work for code optimization.

2. Parameter: the value is set at the beginning of a simulation but remains constant during a single simulation run. In generated Octave models, the literal value of a parameter is not substituted in equations until that equations is evaluated. In future work, parameter equations will be moved outside the time-step loop.
3. Variable: the value is set to an "initial condition" at the beginning of a simulation run and may subsequently change over the whole integration domain. Variables occurring in the form of $der(x)$ (replaced by *der_x* in Octave code) are called *derived state* variables. Depending on the causality a *derived state* variable is assigned (*integral causality* or *derivative causality*), an integration equation or a derivative equation is inserted to solve both of *x* and *der_x*. All other variables are *algebraic* variables. In Octave models, a zero vector of size $numberOfTimeIntervals \times 1$ is created for each variable. The i^{th} element of a vector corresponds to the value of that variable at time step *i*.

5.2.3 Global Variables

The RHS of an ODE, or an algebraic loop, may contain other algebraic variables, such as

```
der(y)=(-1.0)*x+y^2.0
```

in this example. To solve *y* by integration, the value of *x* also need to be known. We can see that from the function definition introduced in section 5.1.1, there is no way to pass the value of *x* into the function body as a formal parameter. The solution to solving this problem is to declare *x* as a *global variable*. In Octave, a variable that has been declared as *global* may be accessed from within a function body without having pass it as a formal parameter.

In generated Octave models, all variables, including constants and parameters, are declared as *global*. Also, there is a reference to the latest evaluated value of each variable at the previous time-step, e.g., *x_last*. These names are also declared as global.

5.2.4 Model Initialization

Before a simulation run, all variables in a model are assigned consistent initial values. This process is called *model initialization*. During this phase, all derivatives, $der(...)$, are treated as unknown algebraic variables. Initial values assigned to variables must be *consistent*. They are subject to the following constrains:

- All equations that are utilized in the intended operation;
- As equations in "initial equation" sections;
- Implicitly by using the value of attribute *start* in the declaration of variables.

Using the dependency graph described in section 4.3.1, it is possible to derive a consistent initial state of a model. Model initialization has not yet been implemented in the μModelica compiler currently. Instead, users are responsible for creating a consistent initial state of a model, by assigning each variable a value at initial time.

5.2.5 Defining Functions

For each explicit ODE, an integration equation is inserted. such an integration equation is turned into a function representing the RHS of the corresponding differential equation in Octave. For example, the ODE

```
der(y)=(-1.0)*x+y^2.0
```

corresponds to the function

```
function y_dot = f_y(yi, ti)
  global x_last;
  global y_last;
  y_dot(1)=(-1.0)*x_last+yi(1)^2.0;
endfunction
```

Functions are also defined for identified algebraic loops. For example, the function

```
function loop1 = f_loop1(x)
  global x_last;
  loop1(1)=(-3.0)+x(2)+x_last+x(1);
  loop1(2)=(-1.0)+x(2)+x(1)*(-2.0);
endfunction
```

is defined to solve the following algebraic loop:

```
Algebraic Loop 1:  -3.0+w+x+z = 0.0
Algebraic Loop 1:  -1.0+w+z*(-2.0) = 0.0
```

5.2.6 The For-Loop

The for-loop is designed to solve variables at each time step, in the order of their dependencies. It starts from the second time step since the value of each variable at the first time step is determined by model initialization. Equations within the for-loop are written in the order of computation dependency, which is the result of sorting described in section 4.3. The reference to the latest evaluated value of a variable is updated right after it is computed at each time step.

ODEs are integrated over the time interval of every two consecutive time steps, i.e., integrated at each $[t(i-1), t(i)]$. Such a time interval is divided into 10 slices, i.e.,

```
time_i=linspace(time(i-1), time(i), 10)'
```

The value of integration at the 10^{th} slice is assigned to the state variable being integrated.

5.2.7 Visualized Output

When the for-loop terminates, the solution signal for each variable is stored in the vector which was created before the simulation run. Octave supports graphical output of simulation results. The following command is used to display solutions graphically

```
plot(time, x)
```

For the above example, this gives figure 5.2

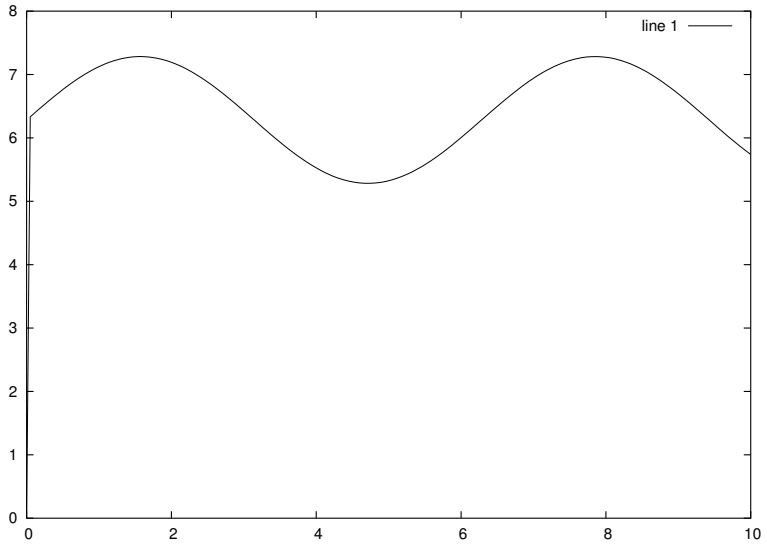

Figure 5.2: GNU Plot Sample

6 Case Study

In order to show that the μModelica compiler is able to solve real problems, a case study is presented in this chapter. The study is based on the simple circuit example mentioned in chapter 1. It is a circuit which consists of two resistors, a capacitor, a sine voltage source, and a ground point. All these components are connected in, as shown in Figure 6.1.

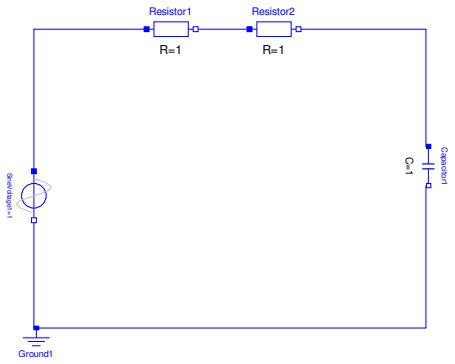

Figure 6.1: An Electrical Circuit

As it will be shown later in this chapter, this is a nontrivial case because the model involves the most important features of Modelica, such as class inheritance, class modifications, components coupled by connection equations, and non-causal modeling with implicit equations. Finally this is a model that ends up with both ODEs and algebraic loops after causality assignment. This chapter presents how the μModelica compiler translates the original Modelica source code of the model into flat Modelica, the transformations of equations, and finally how Octave simulates the model. The simulation result is compared to the one obtained in the demo version of the Modelica commercial tool, *Dymola 5*, by Dynasim AB (`http://www.dynasim.se/`).

6.1 A Modelica Description of the Model

A complete description of the simple circuit model in Modelica is given as follows:

```
// declaring physical quantities
type Voltage = Real (unit="V");

type Current = Real (unit="A");

// define connector class
connector Pin
  Voltage v;
  flow Current i;
end pin;

// define the partial model of components with two pins
partial model TwoPin "Superclass of elements with 2 electrical pins"
  Pin p, n;
  Voltage v;
  Current i;
equation
  v = p.v - n.v;
  0 = p.i + n.i;
  i = p.i;
end TwoPin;

// definition of resistor
model Resistor "Ideal electrical resistor"
  extends TwoPin;
  parameter Real r (unit="Ohm") "Resistance";
equation
  r * i = v;
end Resistor;

// definition of capacitor
model Capacitor "Ideal electrical capacitor"
  extends TwoPin;
  parameter Real c (unit="F") "Capacitance";
equation
  c * der(v) = i;
end Capacitor;

// sine voltage source
model VsourceAC "sin-wave voltage source"
  extends TwoPin;
  parameter Voltage VA = 110 "Amplitude";
  parameter Real f (unit="Hz") = 1 "Frequency";
  constant Real pi = 3.14159265;
equation
  v = VA*sin(2*pi*f*time);
end VsourceAC;
```

```
// the ground point
model Ground "Ground"
  Pin p;
equation
  p.v = 0;
end Ground;

// the complete model
model circuit
  Resistor R1(r=1);
  Resistor R2(r=1);
  Capacitor C(c=1);
  VsourceAC AC;
  Ground G;
equation
  connect (AC.p, R1.p);
  connect (R1.n, R2.p);
  connect (R2.n, C.p);
  connect (C.n, AC.n);
  connect (AC.n, G.p);
end circuit;
```

From the source code, we can see that circuit components are built hierarchically from basic components, i.e., predefined types. Subsequently, these circuit components are connected via *connect statements*.

Since the μModelica compiler currently does not support *import* statements, all the class definitions have to be placed in one file. This file is the input to the compiler.

6.2 Translation to Flat Modelica

Given this file as input, the μModelica compiler eventually generates corresponding Octave code. But there are some intermediate transformation steps which lead to corresponding intermediate representations of the model. This section shows the intermediate representations in the Front End.

During the process of flattening, class inheritance is first expanded. The printout of the intermediate representation of the expanded classes is as follows:

```
// expanded version of Resistor
model Resistor
  Pin p, n;
  Voltage v;
  Current i;
  parameter Real r (unit="Ohm");
equation
  v = p.v - n.v;
  0 = p.i + n.i;
  i = p.i;
  r * i = v;
end Resistor;
```

```
// expanded version of Capacitor
model Capacitor
  extends TwoPin;
  Pin p, n;
  Voltage v;
  Current i;
  parameter Real c (unit="F");
equation
  v = p.v - n.v;
  0 = p.i + n.i;
  i = p.i;
  c * der(v) = i;
end Capacitor;

// expanded version of VsourceAC
model VsourceAC
  Pin p, n;
  Voltage v;
  Current i;
  parameter Voltage VA = 110;
  parameter Real f (unit="Hz") = 1;
  constant Real pi = 3.14159265;
equation
  v = p.v - n.v;
  0 = p.i + n.i;
  i = p.i;
  v = VA*sin(2*pi*f*time);
end VsourceAC;
```

In this model, both `Resistor`, `Capacitor`, and `VsourceAC` are derived classes of `TwoPin`. All declarations and equations in `TwoPin` are copied and inserted into these three classes, respectively. Then the model is translated into flat Modelica in terms of flattening composite components, flattening connect equations, and resolving modifications. After these translation steps have been executed, a flat Modelica description of the model is generated by the Front End:

```
model Circuit
  flow Real  R1_n_i;
  Real  R2_v;
  parameter Real  R1_r
  Real  G_p_v;
  Real  C_n_v;
  Real  R1_p_v;
  flow Real  R1_p_i;
  flow Real  G_p_i;
  Real  R2_i;
  flow Real  AC_n_i;
  parameter Real  R2_r;
```

```
  Real  R1_n_v;
  flow Real  R2_n_i;
  parameter Real  C_c;
  flow Real  R2_p_i;
  parameter Real  AC_VA;
  flow Real  AC_p_i;
  Real  C_i;
  flow Real  C_p_i;
  constant Real  AC_pi;
  Real  R1_v;
  Real  AC_v;
  Real  AC_i;
  Real  C_p_v;
  Real  R1_i;
  Real  C_v;
  Real  AC_p_v;
  Real  AC_n_v;
  Real  R2_p_v;
  flow Real  C_n_i;
  Real  R2_n_v;
  parameter Real  AC_f;
equation
  C_c*der(C_v)=C_i;
  C_v=C_p_v-C_n_v;
  0=C_p_i+C_n_i;
  C_i=C_p_i;
  R1_r*R1_i=R1_v;
  R1_v=R1_p_v-R1_n_v;
  0=R1_p_i+R1_n_i;
  R1_i=R1_p_i;
  G_p_v=0;
  R2_r*R2_i=R2_v;
  R2_v=R2_p_v-R2_n_v;
  0=R2_p_i+R2_n_i;
  R2_i=R2_p_i;
  AC_v=AC_VA*sin(2*AC_f*AC_pi*time);
  AC_v=AC_p_v-AC_n_v;
  0=AC_p_i+AC_n_i;
  AC_i=AC_p_i;
  AC_p_i+R1_p_i=0.0;
  AC_p_v=R1_p_v;
  R1_n_i+R2_p_i=0.0;
  R1_n_v=R2_p_v;
  R2_n_i+C_p_i=0.0;
  R2_n_v=C_p_v;
  G_p_i+AC_n_i+C_n_i=0.0;
  G_p_v=AC_n_v;
  G_p_v=C_n_v;
```

```
end Circuit;
```

In the flat Modelica description of the model, only modifications of real numbers or integers are turned into equations. Modifications of strings, e.g., modification to the `unit` attribute, are ignored.

6.3 Formula Manipulation

The Back End performs formula manipulation on the set of equations declared in the flat Modelica description. Formula manipulation includes the techniques of canonical transformation, causality assignment, sorting and algebraic loop detection. This section shows the result of each transformation step.

6.3.1 Canonical Representation

The set of equations are transformed to the following canonical form:

```
C_c*der(C_v)+C_i*(-1.0)=0.0;
C_n_v+C_p_v*(-1.0)+C_v=0.0;
C_n_i*(-1.0)+C_p_i*(-1.0)=0.0;
C_i+C_p_i*(-1.0)=0.0;
R1_i*R1_r+R1_v*(-1.0)=0.0;
R1_n_v+R1_p_v*(-1.0)+R1_v=0.0;
R1_n_i*(-1.0)+R1_p_i*(-1.0)=0.0;
R1_i+R1_p_i*(-1.0)=0.0;
G_p_v=0.0;
R2_i*R2_r+R2_v*(-1.0)=0.0;
R2_n_v+R2_p_v*(-1.0)+R2_v=0.0;
R2_n_i*(-1.0)+R2_p_i*(-1.0)=0.0;
R2_i+R2_p_i*(-1.0)=0.0;
AC_VA*sin(2*AC_f*AC_pi*time)*(-1.0)+AC_v=0.0;
AC_n_v+AC_p_v*(-1.0)+AC_v=0.0;
AC_n_i*(-1.0)+AC_p_i*(-1.0)=0.0;
AC_i+AC_p_i*(-1.0)=0.0;
AC_p_i+R1_p_i=0.0;
AC_p_v+R1_p_v*(-1.0)=0.0;
R1_n_i+R2_p_i=0.0;
R1_n_v+R2_p_v*(-1.0)=0.0;
C_p_i+R2_n_i=0.0;
C_p_v*(-1.0)+R2_n_v=0.0;
AC_n_i+C_n_i+G_p_i=0.0;
AC_n_v*(-1.0)+G_p_v=0.0;
C_n_v*(-1.0)+G_p_v=0.0;
```

6.3.2 Causality Assignment

The μModelica compiler finds a matching between equations and variables in causality assignment. The result is as follows:

```
Eq: AC_n_i+C_n_i+G_p_i = 0.0
    --is used to solve G_p_i
```

```
Eq: AC_n_v*(-1.0)+G_p_v = 0.0
    --is used to solve AC_n_v
Eq: C_p_i+R2_n_i = 0.0
    --is used to solve C_p_i
Eq: C_n_v*(-1.0)+G_p_v = 0.0
    --is used to solve C_n_v
Eq: R1_n_i+R2_p_i = 0.0
    --is used to solve R1_n_i
Eq: R1_n_v+R2_p_v*(-1.0) = 0.0
    --is used to solve R2_p_v
Eq: R2_n_v+R2_p_v*(-1.0)+R2_v = 0.0
    --is used to solve R2_v
Eq: R2_i*R2_r+R2_v*(-1.0) = 0.0
    --is used to solve R2_i
Eq: R2_i+R2_p_i*(-1.0) = 0.0
    --is used to solve R2_p_i
Eq: R2_n_i*(-1.0)+R2_p_i*(-1.0) = 0.0
    --is used to solve R2_n_i
Eq: AC_n_v+AC_p_v*(-1.0)+AC_v = 0.0
    --is used to solve AC_p_v
Eq: AC_VA*sin(2*AC_f*AC_pi*time)*(-1.0)+AC_v = 0.0
    --is used to solve AC_v
Eq: AC_i+AC_p_i*(-1.0) = 0.0
    --is used to solve AC_i
Eq: AC_n_i*(-1.0)+AC_p_i*(-1.0) = 0.0
    --is used to solve AC_n_i
Eq: AC_p_v+R1_p_v*(-1.0) = 0.0
    --is used to solve R1_p_v
Eq: AC_p_i+R1_p_i = 0.0
    --is used to solve AC_p_i
Eq: C_p_v*(-1.0)+R2_n_v = 0.0
    --is used to solve R2_n_v
Eq: C_c*der(C_v)+C_i*(-1.0) = 0.0
    --is used to solve der(C_v)
Eq: C_n_i*(-1.0)+C_p_i*(-1.0) = 0.0
    --is used to solve C_n_i
Eq: C_n_v+C_p_v*(-1.0)+C_v = 0.0
    --is used to solve C_p_v
Eq: R1_i*R1_r+R1_v*(-1.0) = 0.0
    --is used to solve R1_v
Eq: C_i+C_p_i*(-1.0) = 0.0
    --is used to solve C_i
Eq: R1_n_i*(-1.0)+R1_p_i*(-1.0) = 0.0
    --is used to solve R1_p_i
Eq: R1_n_v+R1_p_v*(-1.0)+R1_v = 0.0
    --is used to solve R1_n_v
Eq: G_p_v = 0.0
    --is used to solve G_p_v
```

```
Eq: R1_i+R1_p_i*(-1.0) = 0.0 is
    --used to solve R1_i
```

By default, the *derived state variable* $der(C\_v)$ is given integral causality. That is, $der(C\_v)$ is treated as an algebraic unknown, and C_v is computed by integration.

6.3.3 Sorting and Algebraic Loop Detection

The μModelica compiler also found that there exists an algebraic dependency loop among some of the equations. This is detected while the equations are sorted into a correct computation order based on their computational dependencies.

```
--------------- Sorting ------------------
(1) R1_p_i---7: R1_n_i*(-1.0)+R1_p_i*(-1.0) = 0.0
(2) R1_i---8: R1_i+R1_p_i*(-1.0) = 0.0
(3) R1_v---5: R1_i*R1_r+R1_v*(-1.0) = 0.0
(4) G_p_v---9: G_p_v = 0.0
(5) AC_n_v---25: AC_n_v*(-1.0)+G_p_v = 0.0
(6) AC_v---14: AC_VA*sin(2*AC_f*AC_pi*time)*(-1.0)+AC_v = 0.0
(7) AC_p_v---15: AC_n_v+AC_p_v*(-1.0)+AC_v = 0.0
(8) R1_p_v---19: AC_p_v+R1_p_v*(-1.0) = 0.0
(9) R1_n_v---6: R1_n_v+R1_p_v*(-1.0)+R1_v = 0.0
(10) R2_p_v---21: R1_n_v+R2_p_v*(-1.0) = 0.0
(11) C_n_v---26: C_n_v*(-1.0)+G_p_v = 0.0
(12) C_p_v---2: C_n_v+C_p_v*(-1.0)+C_v = 0.0
(13) R2_n_v---23: C_p_v*(-1.0)+R2_n_v = 0.0
(14) R2_v---11: R2_n_v+R2_p_v*(-1.0)+R2_v = 0.0
(15) R2_i---10: R2_i*R2_r+R2_v*(-1.0) = 0.0
(16) R2_p_i---13: R2_i+R2_p_i*(-1.0) = 0.0
(17) R1_n_i---20: R1_n_i+R2_p_i = 0.0
(18) AC_p_i---18: AC_p_i+R1_p_i = 0.0
(19) AC_n_i---16: AC_n_i*(-1.0)+AC_p_i*(-1.0) = 0.0
(20) R2_n_i---12: R2_n_i*(-1.0)+R2_p_i*(-1.0) = 0.0
(21) C_p_i---22: C_p_i+R2_n_i = 0.0
(22) C_n_i---3: C_n_i*(-1.0)+C_p_i*(-1.0) = 0.0
(23) G_p_i---24: AC_n_i+C_n_i+G_p_i = 0.0
(24) AC_i---17: AC_i+AC_p_i*(-1.0)= 0.0
(25) C_i---4: C_i+C_p_i*(-1.0) = 0.0
(26) der(C_v)---1: C_c*der(C_v)+C_i*(-1.0) = 0.0

----------- Algebraic Loops --------------
Algebraic Loop: 1
 20: R1_n_i+R2_p_i=0.0
 7: R1_n_i*(-1.0)+R1_p_i*(-1.0)=0.0
 8: R1_i+R1_p_i*(-1.0)=0.0
 5: R1_i*R1_r+R1_v*(-1.0)=0.0
 6: R1_n_v+R1_p_v*(-1.0)+R1_v=0.0
 21: R1_n_v+R2_p_v*(-1.0)=0.0
 11: R2_n_v+R2_p_v*(-1.0)+R2_v=0.0
```

```
 10: R2_i*R2_r+R2_v*(-1.0)=0.0
 13: R2_i+R2_p_i*(-1.0)=0.0
```

6.3.4 Rewriting Equations into Explicit Form

The subset of equations that are involved in an algebraic loop are identified when the whole system is rewritten into explicit form. They are grouped together and are placed in the right position according to the computation order.

```
************* Equations in Explicit Form ***************
G_p_v=0.0
AC_n_v=G_p_v
AC_v=AC_VA*sin(2*AC_f*AC_pi*time)
AC_p_v=AC_n_v+AC_v
R1_p_v=AC_p_v
C_n_v=G_p_v
C_p_v=C_n_v+C_v
R2_n_v=C_p_v
Algebraic Loop 1:  R1_n_i+R2_p_i = 0.0
Algebraic Loop 1:  R1_n_i*(-1.0)+R1_p_i*(-1.0) = 0.0
Algebraic Loop 1:  R1_i+R1_p_i*(-1.0) = 0.0
Algebraic Loop 1:  R1_i*R1_r+R1_v*(-1.0) = 0.0
Algebraic Loop 1:  R1_n_v+R1_p_v*(-1.0)+R1_v = 0.0
Algebraic Loop 1:  R1_n_v+R2_p_v*(-1.0) = 0.0
Algebraic Loop 1:  R2_n_v+R2_p_v*(-1.0)+R2_v = 0.0
Algebraic Loop 1:  R2_i*R2_r+R2_v*(-1.0) = 0.0
Algebraic Loop 1:  R2_i+R2_p_i*(-1.0) = 0.0
AC_p_i=(-1.0)*R1_p_i
AC_n_i=(-1.0)*AC_p_i
R2_n_i=(-1.0)*R2_p_i
C_p_i=(-1.0)*R2_n_i
C_n_i=(-1.0)*C_p_i
G_p_i=(-1.0)*AC_n_i+(-1.0)*C_n_i
AC_i=AC_p_i
C_i=C_p_i
der(C_v)=C_i*C_c^(-1.0)
```

6.4 Octave Code

Finally, the following Octave code is generated for the simple circuit model:

```
# simulation time set up
time_init=input("Please enter initial time: ");
time_end=input("Please enter end time: ");
time_step=input("Please enter time step: ");
num_of_intervals=(time_end - time_init)/time_step;
time = linspace(time_init, time_end, num_of_intervals)';

# Constants
```

```
global AC_pi_last=3.14

# Parameters
global R1_r_last=1
R1_r_last=input("enter parameter value: R1_r(1)");
global R2_r_last=1
R2_r_last=input("enter parameter value: R2_r(1)");
global C_c_last=1
C_c_last=input("enter parameter value: C_c(1)");
global AC_VA_last=110
AC_VA_last=input("enter parameter value: AC_VA(110)");
global AC_f_last=1
AC_f_last=input("enter parameter value: AC_f(1)");

# variables
global C_v=zeros(num_of_intervals, 1);
C_v(1)=input("Please enter initial value of C_v: ");
global C_v_last=C_v(1);
global der_C_v=zeros(num_of_intervals, 1);
der_C_v(1)=input("Please enter initial value of der_C_v: ");
der_C_v_last=der_C_v(1);
global C_i=zeros(num_of_intervals, 1);
C_i(1)=input("Please enter initial value of C_i: ");
global C_i_last=C_i(1);
global AC_i=zeros(num_of_intervals, 1);
AC_i(1)=input("Please enter initial value of AC_i: ");
global AC_i_last=AC_i(1);
global G_p_i=zeros(num_of_intervals, 1);
G_p_i(1)=input("Please enter initial value of G_p_i: ");
global G_p_i_last=G_p_i(1);
global C_n_i=zeros(num_of_intervals, 1);
C_n_i(1)=input("Please enter initial value of C_n_i: ");
global C_n_i_last=C_n_i(1);
global C_p_i=zeros(num_of_intervals, 1);
C_p_i(1)=input("Please enter initial value of C_p_i: ");
global C_p_i_last=C_p_i(1);
global R2_n_i=zeros(num_of_intervals, 1);
R2_n_i(1)=input("Please enter initial value of R2_n_i: ");
global R2_n_i_last=R2_n_i(1);
global AC_n_i=zeros(num_of_intervals, 1);
AC_n_i(1)=input("Please enter initial value of AC_n_i: ");
global AC_n_i_last=AC_n_i(1);
global AC_p_i=zeros(num_of_intervals, 1);
AC_p_i(1)=input("Please enter initial value of AC_p_i: ");
global AC_p_i_last=AC_p_i(1);
global R1_n_i=zeros(num_of_intervals, 1);
R1_n_i(1)=input("Please enter initial value of R1_n_i: ");
global R1_n_i_last=R1_n_i(1);
```

```
global R1_p_i=zeros(num_of_intervals, 1);
R1_p_i(1)=input("Please enter initial value of R1_p_i: ");
global R1_p_i_last=R1_p_i(1);
global R1_i=zeros(num_of_intervals, 1);
R1_i(1)=input("Please enter initial value of R1_i: ");
global R1_i_last=R1_i(1);
global R1_v=zeros(num_of_intervals, 1);
R1_v(1)=input("Please enter initial value of R1_v: ");
global R1_v_last=R1_v(1);
global R1_n_v=zeros(num_of_intervals, 1);
R1_n_v(1)=input("Please enter initial value of R1_n_v: ");
global R1_n_v_last=R1_n_v(1);
global R2_p_v=zeros(num_of_intervals, 1);
R2_p_v(1)=input("Please enter initial value of R2_p_v: ");
global R2_p_v_last=R2_p_v(1);
global R2_v=zeros(num_of_intervals, 1);
R2_v(1)=input("Please enter initial value of R2_v: ");
global R2_v_last=R2_v(1);
global R2_i=zeros(num_of_intervals, 1);
R2_i(1)=input("Please enter initial value of R2_i: ");
global R2_i_last=R2_i(1);
global R2_p_i=zeros(num_of_intervals, 1);
R2_p_i(1)=input("Please enter initial value of R2_p_i: ");
global R2_p_i_last=R2_p_i(1);
global R2_n_v=zeros(num_of_intervals, 1);
R2_n_v(1)=input("Please enter initial value of R2_n_v: ");
global R2_n_v_last=R2_n_v(1);
global C_p_v=zeros(num_of_intervals, 1);
C_p_v(1)=input("Please enter initial value of C_p_v: ");
global C_p_v_last=C_p_v(1);
global C_n_v=zeros(num_of_intervals, 1);
C_n_v(1)=input("Please enter initial value of C_n_v: ");
global C_n_v_last=C_n_v(1);
global R1_p_v=zeros(num_of_intervals, 1);
R1_p_v(1)=input("Please enter initial value of R1_p_v: ");
global R1_p_v_last=R1_p_v(1);
global AC_p_v=zeros(num_of_intervals, 1);
AC_p_v(1)=input("Please enter initial value of AC_p_v: ");
global AC_p_v_last=AC_p_v(1);
global AC_v=zeros(num_of_intervals, 1);
AC_v(1)=input("Please enter initial value of AC_v: ");
global AC_v_last=AC_v(1);
global AC_n_v=zeros(num_of_intervals, 1);
AC_n_v(1)=input("Please enter initial value of AC_n_v: ");
global AC_n_v_last=AC_n_v(1);
global G_p_v=zeros(num_of_intervals, 1);
G_p_v(1)=input("Please enter initial value of G_p_v: ");
global G_p_v_last=G_p_v(1);
```

```
# functions to compute ODEs
function C_v_dot = f_C_v(C_vi, ti)
  global C_i_last;
  global C_c_last;
  C_v_dot(1)=C_i_last*C_c_lastˆ(-1.0);
endfunction

# functions to compute algebraic loops
function loop1 = f_loop1(x)
  global R1_r_last;
  global R1_p_v_last;
  global R2_n_v_last;
  global R2_r_last;
  loop1(1)=x(1)+x(9);
  loop1(2)=x(1)*(-1.0)+x(2)*(-1.0);
  loop1(3)=x(3)+x(2)*(-1.0);
  loop1(4)=x(3)*R1_r_last+x(4)*(-1.0);
  loop1(5)=x(5)+R1_p_v_last*(-1.0)+x(4);
  loop1(6)=x(5)+x(6)*(-1.0);
  loop1(7)=R2_n_v_last+x(6)*(-1.0)+x(7);
  loop1(8)=x(8)*R2_r_last+x(7)*(-1.0);
  loop1(9)=x(8)+x(9)*(-1.0);
endfunction

for i=2:num_of_intervals
  # Equation 1
  G_p_v(i)=0.0;
  G_p_v_last=G_p_v(i);
  # Equation 2
  AC_n_v(i)=G_p_v_last;
  AC_n_v_last=AC_n_v(i);
  # Equation 3
  AC_v(i)=AC_VA_last*sin(2*AC_f_last*AC_pi_last*time(i));
  AC_v_last=AC_v(i);
  # Equation 4
  AC_p_v(i)=AC_n_v_last+AC_v_last;
  AC_p_v_last=AC_p_v(i);
  # Equation 5
  R1_p_v(i)=AC_p_v_last;
  R1_p_v_last=R1_p_v(i);
  # Equation 6
  C_n_v(i)=G_p_v_last;
  C_n_v_last=C_n_v(i);
  # Equation 7
  C_p_v(i)=C_n_v_last+C_v_last;
  C_p_v_last=C_p_v(i);
  # Equation 8
```

```
R2_n_v(i)=C_p_v_last;
R2_n_v_last=R2_n_v(i);
# Equation 9
init_guess=zeros(9, 1);
[loop1, info]=fsolve("f_loop1", init_guess);
R1_n_i(i)=loop1(1);
R1_n_i_last=R1_n_i(i);
R1_p_i(i)=loop1(2);
R1_p_i_last=R1_p_i(i);
R1_i(i)=loop1(3);
R1_i_last=R1_i(i);
R1_v(i)=loop1(4);
R1_v_last=R1_v(i);
R1_n_v(i)=loop1(5);
R1_n_v_last=R1_n_v(i);
R2_p_v(i)=loop1(6);
R2_p_v_last=R2_p_v(i);
R2_v(i)=loop1(7);
R2_v_last=R2_v(i);
R2_i(i)=loop1(8);
R2_i_last=R2_i(i);
R2_p_i(i)=loop1(9);
R2_p_i_last=R2_p_i(i);
# Equation 10
AC_p_i(i)=(-1.0)*R1_p_i_last;
AC_p_i_last=AC_p_i(i);
# Equation 11
AC_n_i(i)=(-1.0)*AC_p_i_last;
AC_n_i_last=AC_n_i(i);
# Equation 12
R2_n_i(i)=(-1.0)*R2_p_i_last;
R2_n_i_last=R2_n_i(i);
# Equation 13
C_p_i(i)=(-1.0)*R2_n_i_last;
C_p_i_last=C_p_i(i);
# Equation 14
C_n_i(i)=(-1.0)*C_p_i_last;
C_n_i_last=C_n_i(i);
# Equation 15
G_p_i(i)=(-1.0)*AC_n_i_last+(-1.0)*C_n_i_last;
G_p_i_last=G_p_i(i);
# Equation 16
AC_i(i)=AC_p_i_last;
AC_i_last=AC_i(i);
# Equation 17
C_i(i)=C_p_i_last;
C_i_last=C_i(i);
# Equation 18
```

```
    der_C_v(i)=C_i_last*C_c_last^(-1.0);
    der_C_v_last=der_C_v(i);
    time_i=linspace(time(i-1), time(i), 10)';
    C_vi0=[C_v_last];
    C_v_i=lsode("f_C_v", C_vi0, time_i);
    C_v(i)=C_v_i(10);
    C_v_last=C_v(i);
  endfor
```

6.5 Simulation Result

Given the following initial setup of a simulation run:

```
initial time: 0
end time: 10
time step: 0.02

parameter R1_r=1.0
parameter R2_r=1.0
parameter C_c=1.0
parameter AC_VA=110.0
parameter AC_f=1.0

initial value of all variables: 0
```

Octave generates the *C_v* signal, as shown in Figure 6.2.

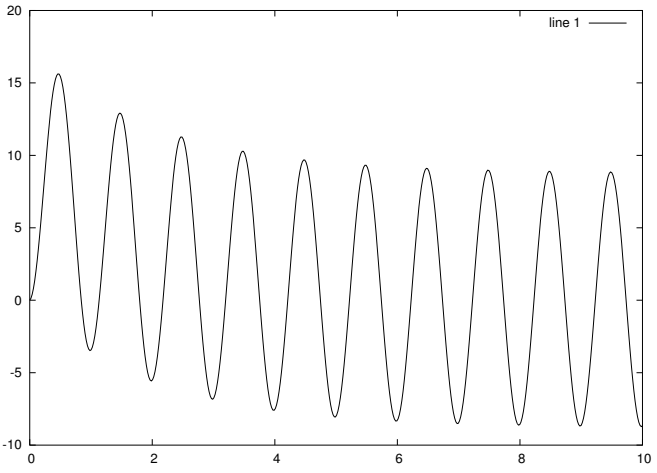

Figure 6.2: *C_v* produced by the *μ*Modelica Compiler and Octave

Given the same model as input to the demo version of the commercial tool Dymola, and the same initial simulation setup, Dymola generates the *C_v* signal shown in Figure 6.3.

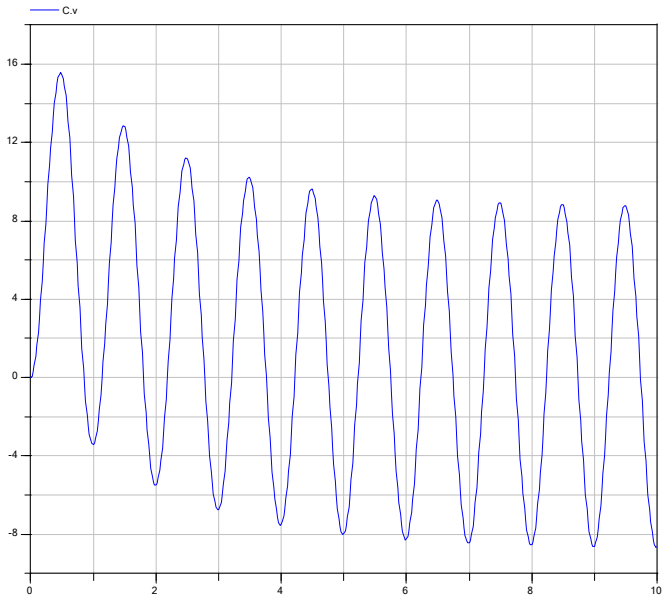

Figure 6.3: *C_v* produced by the Demo version of Dymola

Also, comparisons are made between the *C_i* signals generated by the two tools, as well as *AC_i*. These variables are shown in Figure 6.4, Figure 6.5, Figure 6.6, and Figure 6.7.

6.6 Conclusion

We can see that the simulation result given by the *μ*Modelica Compiler and Octave is almost identical to that given by the demo version of Dymola. Even though large-scale testing has not yet been performed, this case study shows that the compiler is able to compile and simulate non-trivial models of continuous system.

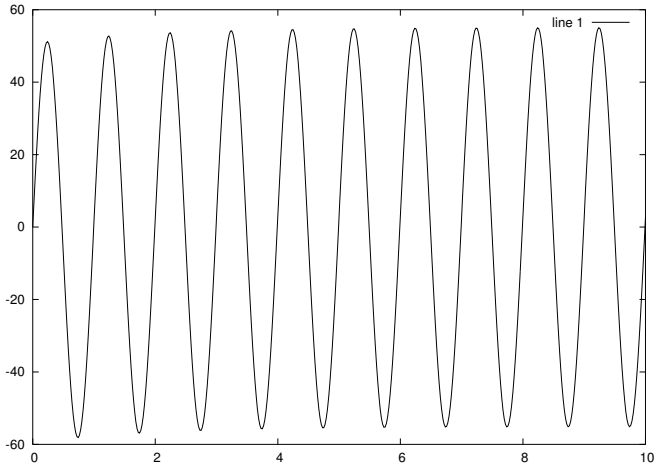

Figure 6.4: *C_i* produced by the μModelica Compiler and Octave

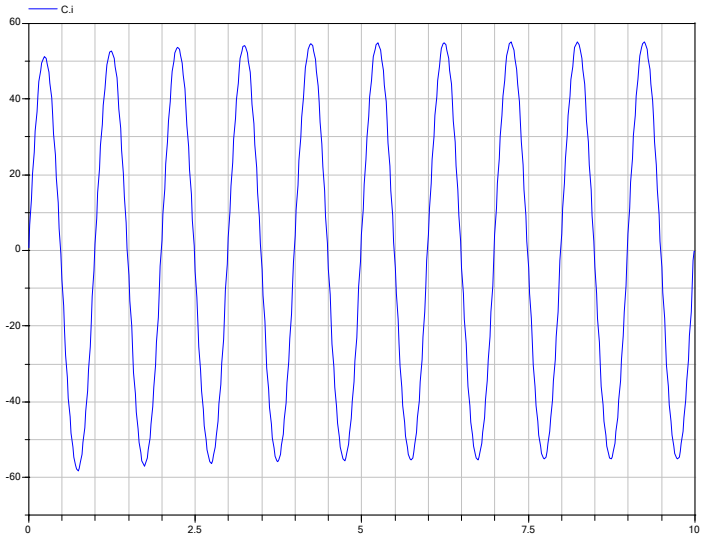

Figure 6.5: *C_i* produced by the Demo version of Dymola

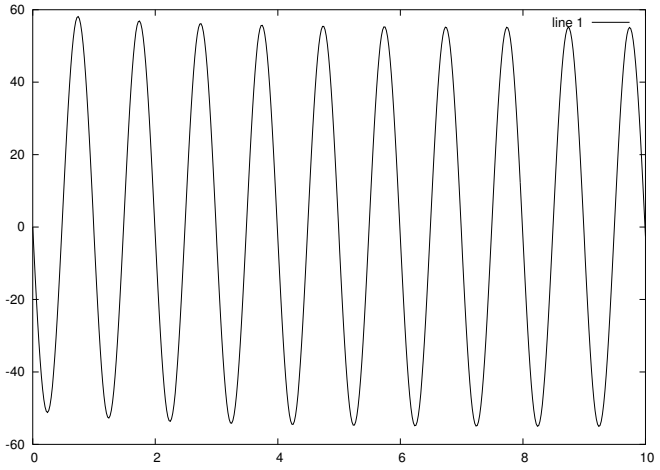

Figure 6.6: *AC_i* produced by the μModelica Compiler and Octave

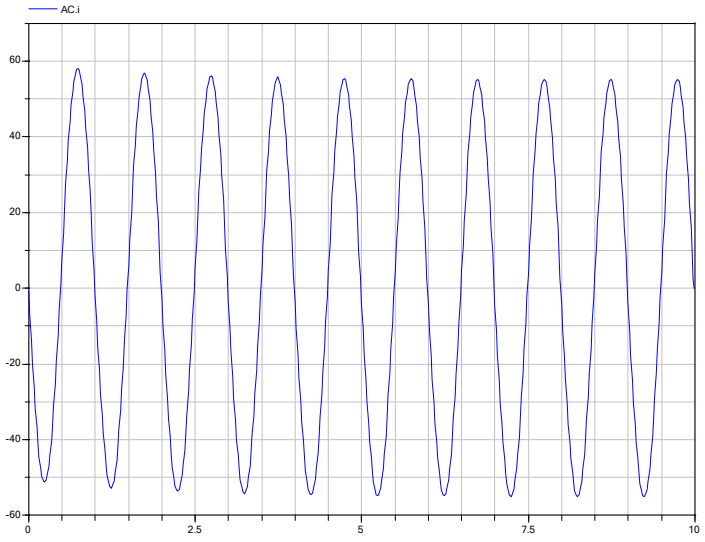

Figure 6.7: *AC_i* produced by the Demo version of Dymola

7 Future Work

Given the objective that we are to build a research prototype compiler for Modelica, the μModelica compiler currently only focuses on a subset of Modelica, which is the real essence of the language—non-causal modeling. As shown in chapter 6, the compiler is able to solve non-trivial problems. But compiling and simulating large models is not possible due to the absence of support for some language features, such as import statements, arrays and matrices, etc. In order to make it possible, and to employ large scale testing on the Modelica standard library, more language features will be implemented in the μModelica compiler. Also, as a research prototype compiler, we are interested in implementing some advanced formula manipulation techniques, such as *teaing* for solving algebraic loops [18], and *inline integration* [12]. This chapter gives an introduction to some of these techniques, and proposes the future work for the μModelica compiler.

7.1 More Language Features

Among the language features to be supported, resolving *import statements* and supporting *arrays* are the most important ones. With the support of these two features, we can make use of the Modelica standard library. Therefore, we will be able to simulate large models and perform large scale testing, which will in turn give us feedback to improve the design and implementation of the compiler.

7.1.1 Import Statement

An *import Statement* is introduced by the following *import clause*:

```
import (IDENT "=" name | name ["." "*"])
```

It can be either a qualified import statement, e.g., **import** `A.B.C`, and **import** `D=A.B.C`, or an unqualified import statement, e.g., **import** `A.B.*`. The following example demonstrates various forms of import statement:

```
package A
  package B
    partial model C
      Real x;
    end C;
    model D
      extends C(x=5);
    end D;
  end B;
  package B1
```

```
    model C
      extends B.C(x=4);
    end C;
  end B1;
  package B2
    model C
      extends B.C(x=7);
    end C;
    model E=B.C(x=6);
    model F=B.C(x=10);
  end B2;
end A;

class Import1
  import A.B.*;
  import A.B2.*;
  import A.B1.C;
  import MyC=A.B2.C; //Note that a qualified import takes
                     // precedence over a unqualified import
  C c;
  D d;
  E e;
  MyC myc;
end Import1;
```

Qualified import statements may only import a package or an element of a package. For example, in **import** `A.B.C`, or **import** `D=A.B.C`, `A.B` must be a package, while `C` can either be a package or an element of a package. Unqualified import statements may only import elements from packages, e.g., in **import** `A.B.*`, `A.B` must be a package.

Lookup of the name in an import statement is different from the normal lexical lookup. The first part of the name, e.g., `A` in `A.B`, is looked up at the top level.

Classes imported from external files can be loaded in two different ways. One of them is the *pessimistic* approach, that is, whenever an import statement is resolved, all imported classes are loaded. Another approach is the *optimistic* approach, that is, a imported element will not be loaded until it is used. For instance, in the sample model, class `A.B2.F` will not be loaded because it is not used in class `Import1`. The first approach is easier to implement. But the latter one is more efficient and use less memory space.

7.1.2 Arrays

Modelica supports *arrays* and *matrices*. An array variable can be declared by appending dimensions after the type-specifier name or after a component name. For example

```
model Array1
  Integer x[5] = {1,2,3,4,5};
  Integer[3] y = 1:3;
end Array1;
```

declares two arrays: `x` of size 5, and `y` of size 3. The flat Modelica description of this model is as follows:

```
class Array1
  Integer x[1];
  Integer x[2];
  Integer x[3];
  Integer x[4];
  Integer x[5];
  Integer y[1];
  Integer y[2];
  Integer y[3];
equation
  x[1] = 1;
  x[2] = 2;
  x[3] = 3;
  x[4] = 4;
  x[5] = 5;
  y[1] = (1:3)[1];
  y[2] = (1:3)[2];
  y[3] = (1:3)[3];
end Array1;
```

From this description, we can see that the semantics of array variables is defined by expanding arrays to individual scalar variables. The implementation of arrays is related to the implementation of relevant language features such as the *for-loop* construct. Further studies on arrays will be carried out in the near future.

7.2 Formula Manipulation Techniques

This section introduces some of the formula manipulation techniques we have studied, which are important in improving simulation run-time efficiency. These techniques will be implemented in the future version of the μModelica compiler.

7.2.1 Eliminate Aliases

Recall that the flat Modelica description of the simple circuit model includes the following set of equations:

```
C_c*der(C_v)=C_i;
C_v=C_p_v-C_n_v;
0=C_p_i+C_n_i;
C_i=C_p_i;
R1_r*R1_i=R1_v;
R1_v=R1_p_v-R1_n_v;
0=R1_p_i+R1_n_i;
R1_i=R1_p_i;
G_p_v=0;
R2_r*R2_i=R2_v;
```

```
R2_v=R2_p_v-R2_n_v;
0=R2_p_i+R2_n_i;
R2_i=R2_p_i;
AC_v=AC_VA*sin(2*AC_f*AC_pi*time);
AC_v=AC_p_v-AC_n_v;
0=AC_p_i+AC_n_i;
AC_i=AC_p_i;
AC_p_i+R1_p_i=0.0;
AC_p_v=R1_p_v;
R1_n_i+R2_p_i=0.0;
R1_n_v=R2_p_v;
R2_n_i+C_p_i=0.0;
R2_n_v=C_p_v;
G_p_i+AC_n_i+C_n_i=0.0;
G_p_v=AC_n_v;
G_p_v=C_n_v;
```

Due to the "connect" statements in coupled models, the original set of equations contains many *aliases*, e.g., there exist many trivial equations of the type $a = b$ or $a + b = 0$. Actually they are the same variables stored under different names. In the simple circuit example, we can see that 16 out of 26 equations are of this type. It will seriously affect the simulation run-time efficiency if all these trivial equations are computed during the simulation process. Actually, some of these equations can be eliminated without affecting the simulation result. This section presents how these equations can be eliminated.

The algorithm for eliminating aliases is straightforward. We only need to get rid of the equations of the type $a = b$, and replace all occurrences of variable a in all other equations by variable b. Also, this rule applies to the following variants of $a = b$:

- $a = -b$
- $-a = b$
- $-a = -b$
- $a + b = 0$
- $a - b = 0$
- $-a + b = 0$
- $-a - b = 0$
- either a or b is a constant

For example, equations of the type $a + b = 0$ is eliminated and all occurrences of a are replaced by -b.

There is an exception to this rule: variables that were declared as **input** or **output** should *not* be eliminated. For instance, if a is an input or output variable, the equation $a = b$ will be eliminated as well, but all occurrences of b are replaced by a. If both a and b are declared as input or output variables, the equation will not be eliminated.

The eliminated variables are no longer visible to the simulator. They will not be computed at simulation run-time. But a user may be interested in knowing the simulation output of some of those eliminated variables. This problem can be solved by keeping a reference table which stores

the relationship of the eliminated variables to variables computed at simulation run-time. These variables will only be computed when required.

7.2.2 Tearing

Once equations are identified as forming an algebraic loop, they are isolated and will be solved simultaneously, either with a symbolic solver or a numerical solver. There are two types of algebraic loops, linear algebraic loops and non-linear algebraic loops. Linear algebraic loops can be solved analytically using Cramer's rule, or with numerical techniques in case the analytical solution grows too large. Non-linear algebraic loops can not generally be solved by formula manipulation. It may be preferrable to employ a numerical method to solve such a set of equations.

The technique to solve non-linear algebraic loops we are to discuss here is called *tearing*, which was introduced by Kron in 1962 [18]. It is a simple technique to reduce a large system of linear or non-linear algebraic equations to a smaller system of equations. It consists of finding a reduced subset of variables over which to iterate, so that the remaining paired variables can be calculated explicitly as a function of these variables.

Consider a set of non-linear algebraically coupled equations $\mathbf{h}$ to be solved for the unknown vector $\mathbf{z}$:

$$\mathbf{0} = \mathbf{h}(\mathbf{z}) \tag{7.1}$$

Tearing means breaking algebraic loops in the dependency structure of equations and variables. A subset of $\mathbf{z}$, called $\mathbf{z_1}$, are chosen as *tearing variables*. A subset of $\mathbf{h}$, called $\mathbf{h_1}$, are chosen as *residual equations*. The choice is made in such a way that the remainder of $\mathbf{z}$, called $\mathbf{z_2}$, can be calculated in sequence using the remaining equations $\mathbf{h_2}$, assuming that the $\mathbf{z_1}$ variables are known, i.e.:

$$\mathbf{z_2} = \mathbf{h_2}(\mathbf{z_1}) \qquad \mathbf{0} = \mathbf{h_1}(\mathbf{z_1}, \mathbf{z_2}) \tag{7.2}$$

This system of equations can be solved by Newton iteration over the tearing variables $\mathbf{z_1}$. The numerical procedure to compute $\mathbf{z}$ is as follow:

- Choose $\mathbf{z_1}$
- Give an estimate to $\mathbf{z_1}$
- Compute : $\mathbf{z_2} = \mathbf{h_2}(\mathbf{z_1})$
- Compute the residual in $res(\mathbf{z_1}) = \mathbf{h_1}(\mathbf{z_1}, \mathbf{z_2})$
- Iterate until $res(\mathbf{z_1})$ are within tolerance.

We can observe from this procedure that it reduces the dimension of the iterated system of equations from $dim(h) = dim(h_1) + dim(h_2)$ down to $dim(h_1)$.

However, the optimal selection of tearing variables and residual equations is not trivial. This is because:

- The more tearing variables there are, the greater the computational overhead.
- Numerical errors may differ considerably from one selection to another.
- Fewer tearing variables may mean greater errors since the errors are propagated through the equations and may be amplified.

These factors make it almost impossible to know automatically whether a selection of tearing variables is good or not. This means that, in general, it is preferable for the user to make the choice based on knowledge of the problem domain. But the compiler itself has to check whether the user's selection is valid.

7.2.3 Inline Integration

Inline integration is a new method for solving DAEs using a mixed symbolic and numerical approach, which is proposed by [12].

In practice, it is either the modeling software or simulation software that converts the continuous-time problem to a discrete-time problem that can be solved through iteration. Traditionally this task was assigned to simulators. However, the concept *inline integration* enables the conversion of continuous-time problem to discrete-time problem at compile time by modeling software.

The basic idea of inline integration is to transform ODEs to algebraic equations through either an explicit or implicit integration method. The original set of DAEs will then be converted into a set of purely algebraic equations. With this technique, a compiler is able to generate more efficient simulation run-time code.

Continuous-time systems can essentially be represented as state-space models through a set of ODEs:

$$der(\mathbf{x}) = \mathbf{f}(\mathbf{x}, t); \qquad \mathbf{x}(t_0) = \mathbf{x}_0 \tag{7.3}$$

where $der()$ denotes the time derivative, $\mathbf{x}$ is the vector of state variables, t denotes time, and $\mathbf{f}$ is a set of assignment statements specifying how the derivatives of $\mathbf{x}$ are computed, assuming the state variables $\mathbf{x}$ are known.

Solving (7.3) by any explicit integration method is straightforward. In the forward Euler method, the derivative of the state vector $\mathbf{x}$ is approximated by:

$$der(\mathbf{x}(t_n)) = der(\mathbf{x}_n) = \frac{\mathbf{x}_{n+1} - \mathbf{x}_n}{h} \tag{7.4}$$

where $\mathbf{x}_{n+1} = \mathbf{x}(t_{n+1})$ is the unknown value of $\mathbf{x}$ at the new time instant $t_{n+1} = t_n + h$, $\mathbf{x}_n = \mathbf{x}(t_n)$ is the known value of $\mathbf{x}$ at the previous time instant t_n, and h is the time increment. Substituting $der(\mathbf{x})$ in (7.4) by (7.3) leads to the following recursive formula:

$$\mathbf{x}_{n+1} = \mathbf{x}_n + h * \mathbf{f}(\mathbf{x}_n, t_n); \qquad \mathbf{x}_0 = \mathbf{x}(t_0) \tag{7.5}$$

This formula can be used to solve the ODE, and it works well for non-stiff systems.

But unfortunately, explicit integration methods are not well suited for stiff systems or systems which contain algebraic loops. Implicit integration methods are more appropriate in such cases [12]. Using the backward Euler method, the derivative of the state vector $\mathbf{x}$ is approximated by:

$$der(\mathbf{x}_{n+1}) = \frac{\mathbf{x}_{n+1} - \mathbf{x}_n}{h} \tag{7.6}$$

Substituting $der(\mathbf{x}_{n+1})$ in (7.6) by (7.3) leads to

$$\mathbf{x}_{n+1} = \mathbf{x}_n + h * \mathbf{f}(\mathbf{x}_{n+1}, t_{n+1}) \tag{7.7}$$

where $\mathbf{x}_{n+1}$ is the unknown to be solved, given $\mathbf{x}_n$ and t_{n+1}. Equation (7.7) can be rewritten as :

$$\mathbf{x} = old(\mathbf{x}) + h * der(\mathbf{x}) \tag{7.8}$$

In general, (7.8) is a non-linear equation for $\mathbf{x}_{n+1}$, and usually it will be solved by numerical methods.

Adding equation (7.5) (for non-stiff systems), or (7.8) (for stiff systems or systems containing algebraic loops) to the original model represented by (7.3) will transform the the system to a set

of purely algebraic equations, assuming that both $der(\mathbf{x})$ and $\mathbf{x}$ are unknowns. Given the following sets of equations:

$$\begin{cases} AC\_v &= sin(time) \\ C\_i &= R1\_v * R1\_r^{-1.0} \\ R2\_v &= R2\_r * C\_i \\ R2\_p\_v &= R2\_v + (-1.0) * C\_v \\ R1\_v &= AC\_v + (-1.0) * R2\_p\_v \\ der(C\_v) &= C\_i * C\_c^{-1.0} \end{cases} .$$

adding (7.8) leads to the following system:

$$\begin{cases} AC\_v &= sin(time) \\ C\_i &= R1\_v * R1\_r^{-1.0} \\ R2\_v &= R2\_r * C\_i \\ R2\_p\_v &= R2\_v + (-1.0) * C\_v \\ R1\_v &= AC\_v + (-1.0) * R2\_p\_v \\ der(C\_v) &= C\_i * C\_c^{-1.0} \\ C\_v &= h * der(C\_v) + old(C\_v) \end{cases} .$$

where both $der(C\_v)$ and C_v are treated as algebraic variables. This set of equations need to be sorted again and checked for algebraic loops. To solve the system, a simulator no longer needs to have an integrator. It only needs a solver for algebraic loops.

In many cases a model contains algebraic loops, and implicit inline integration method may introduce extra algebraic loops. The tearing technique can be combined with inline integration to solve the algebraic loop problem elegantly. It allows the automated transformation of model equations to their discretized form in a simple way. Assuming the model is specified in ODE form by (7.3), tearing provides an elegant formulation:

$$der(\mathbf{x}) = \mathbf{f}(\mathbf{x}, t) \tag{7.9}$$

$$\mathbf{x} = old(\mathbf{x}) + h * der(\mathbf{x}) + res(\mathbf{x}) \tag{7.10}$$

In this discretized model, the original equations are not changed, while additional discretization equations are added. Here both $\mathbf{x}$ and $der(\mathbf{x})$ are considered as unknowns. $\mathbf{x}$ is selected as tearing variables. The solver will give an estimate for $\mathbf{x}$, then $der(\mathbf{x})$ is computed by the state equation (7.9). Finally, the residuals are computed via equation (7.10) and returned to the solver. This process is iterated until converged.

A general algorithm that transforms a DAE down to a suitable discretized form in an automatic manner has been developed in [12]. A system of DAE is represented as

$$\mathbf{0} = \mathbf{f}(der(\mathbf{x}), \mathbf{x}, \mathbf{w}, t); \qquad \mathbf{x}(t_0) = \mathbf{x}_0 \tag{7.11}$$

where $\mathbf{x}$ is, as mentioned before, the vector of unknown variables that appear in the model in differentiated form, whereas $\mathbf{w}$ is the vector of unknown purely algebraic variables. The algorithm includes the following steps to perform inline integration of a DAE system:

1. Transform the system to casual form, sort the equations, and check if there exist algebraic loops, assuming that $\mathbf{x}$ is known, and that $\mathbf{w}$ and der($\mathbf{x}$) are unknown.
2. For every x_i that can be solved explicitly in the partitioned equations, add the following equation
$$x_i = h * der(x_i) + old(x_i) + res(x_i) \tag{7.12}$$
For all other x_j, add the same equation but without the term $res(x_j)$.

3. If the assigned equation of $der(x_j)$ or w_k appears in an algebraic loop, add the term $res(der(x_j))$ or $res(w_k)$ to the corresponding model equation.
4. Repeat the first step while **w**, der(**x**) and **x** are all treated as unknown variables, thereby utilizing the tearing information. As a result, nonlinear, discretized model equations are produced.

7.2.4 Higher Index Problem

Mathematical non-causal modeling of physical systems may result in *higher index* DAEs. However, there are no general purpose solvers for higher index DAEs. These systems are usually solved in terms of index reduction as described by Pantelides [21]. This topic is beyond the scope this thesis.

Bibliography

[1] Python 2.2.3 documentation, May 2003. `http://www.python.org/doc/2.2.3/`.

[2] Alfred Aho, Ravi Sethi, and Jeffrey Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[3] Andrew W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 1997.

[4] Modelica Association. Modelica - a unified object-oriented language for physical systems modeling, tutorial, version 1.4. December 2000.

[5] Modelica Association. Modelica - a unified object-oriented language for physical systems modeling, language specification, version 2.0. July 2002.

[6] Peter Bunus and Peter Fritzson. Automated static analysis of equation-based components. *Simulation*, 80:321 – 345, August 2004.

[7] F.E. Cellier and H. Elmqvist. Automated formula manipulation supports object-oriented continuous-system modeling. *IEEE Control Systems*, 1(1), September 1993.

[8] François E. Cellier. *Continuous System Modeling*. Springer-Verlag, New York, 1991.

[9] E.A. Dinic. Algorithm for solution of a problem of maximum flow in a network with power estimation. *Soviet Math. Dokl.*, 11:1277–1280, 1970.

[10] I. S. Duff and J. K. Reid. Algorithm 529: Permutations to block triangular form [f1]. *ACM Transactions on Mathematical Software (TOMS)*, 4(2):189–192, 1978.

[11] I. S. Duff and J. K. Reid. An implementation of tarjan's algorithm for the block triangularization of a matrix. *ACM Transactions on Mathematical Software (TOMS)*, 4(2):137–147, 1978.

[12] Hilding Elmqvist, Martin Otter, and Francois E. Cellier. Inline integration: A new mixed symbolic /numeric approach for solving differential– algebraic equation systems. 1995.

[13] S. Even and Robert Endre Tarjan. Network flow and testing graph connectivity. *SIAM J. Comput.*, 4:507–518, 1975.

[14] G. Fabian, D.A. van Beek, and J.E. Rooda. Index reduction and discontinuity handling using substitute equations. *Mathematical and Computer Modelling of Dynamical Systems*, 7(2):173–187, 1 2001.

[15] Peter Fritzson. *Principles of Object-Oriented Modelling and Simulaation with Modelica 2.1*. Wiley Inter-Science, 2004.

[16] Peter Fritzson and Vadim Engelson. Modelica—a unified object-oriented language for system moedling and simulation. page 19, July 1998.

[17] EA International. Ecosimpro mathematical algorithms. December 1999.

[18] G Kron. Diakoptics—the piecewise solution of large-scale systems. 1962.

[19] John R. Levine, Tony Mason, and Doug Brown. *lex & yacc*. O'Reilly & Associates, second edition, October 1992.

[20] Martin Otter and Pieter Mosterman. The DSblock model interface, version 4.0. Technical report, Modelica Design, 1999.

[21] C.C. Pantelides. The consistent initialization of differential-algebraic systems. *SIAM Journal on Statistical and Scientific Computing*, 9(2):213–231, 1 1988.

[22] Linda R. Petzold. A description of DASSL: A differential/algebraic system solver. Technical Report SAND82-8637, Sandia National Laboratories, Livermore, California, 1982.

[23] Simulink. *Using Simulink*. The MathWorks, Natick, MA, June 2004.

[24] Jon C. Strauss, Donald C. Augustin, Mark S. Fineberg, Bruce B. Johnson, Robert N. Linebarger, and F. John Sansom. The SCi Continuous System Simulation Language (CSSL). *Simulation*, 9(6):281 – 303, December 1967.

[25] Michael Tiller. *Introduction to Physical Modeling with Modelica*, volume 615 of *International Series in Engineering and Science*. Kluwer Academic, May 2001.

[26] Hans Vangheluwe, Bhama Sridharan, and Indrani A.V. An algorithm to implement a canonical representation of algebraic expressions and equations in atom^3. April 2003.

[27] Henk Vanhooren, Jurgen Meirlaen, Youri Amerlinck, Filip Claeys, Hans L. Vangheluwe, and Peter A. Vanrolleghem. WEST: Modelling biological wastewater treatment. *Journal of Hydroinformatics*, 5(1):27–50, 2003.

A Grammar

A.1 Stored definition

```
stored_definition:
  [ within [ name ] ";" ]
  { [ final ] class\_definition ";" }
```

A.2 Class Definition

```
class_definition :
  [ encapsulated ]
  [ partial ]
  ( class | model | record | block | connector | type | package | function )
 IDENT class_specifier

class_specifier :
  string_comment composition end IDENT
  | "=" base_prefix name [ array_subscripts ] [ class_modification ] comment
  | "=" enumeration "(" [enum_list] ")" comment

base_prefix :
  type_prefix

enum_list : enumeration_literal { "," enumeration_literal}
enumeration_literal : IDENT comment

composition :
  element_list
  { public element_list |
    protected element_list |
    equation_clause |
    algorithm_clause
   }
  [ external [ language_specification ]
                  [ external_function_call ] ";" [ annotation ";" ] ]

language_specification :
  STRING
```

```
external_function_call :
  [ component_reference "=" ]
  IDENT "(" [ expression { "," expression } ] ")"

element_list :
  { element ";" | annotation ";" }

element :
  import_clause |
  extends_clause |
  [ final ]
  [ inner | outer ]
  ( ( class_definition | component_clause) |
    replaceable ( class_definition | component_clause)
     [constraining_clause comment])

import_clause :
  import ( IDENT "=" name | name ["." "*"] ) comment
```

A.3 Extends

```
extends_clause :
  extends name [ class_modification ]

constraining_clause :
  extends_clause
```

A.4 Component Clause

```
component_clause:
  type_prefix type_specifier [ array_subscripts ] component_list

type_prefix :
  [ flow ] [ discrete | parameter | constant ] [ input | output ]

type_specifier :
  name

component_list :
  component_declaration { "," component_declaration }

component_declaration :
  declaration comment

declaration :
  IDENT [ array_subscripts ] [ modification ]
```

A.5 Modification

```
modification :
  class_modification [ "=" expression ]
  | "=" expression
  | ":=" expression

class_modification :
  "(" [ argument_list ] ")"

argument_list :
  argument { "," argument }

argument :
  element_modification
  | element_redeclaration

element_modification :
  [ each ] [ final ] component_reference modification string_comment

element_redeclaration :
  redeclare [ each ] [ final ]
  ( ( class_definition | component_clause1) |
    replaceable ( class_definition | component_clause1)
      [constraining_clause])

component_clause1 :
  type_prefix type_specifier component_declaration
```

A.6 Equations

```
equation_clause :
  [ initial ] equation { equation ";" | annotation ";" }

algorithm_clause :
  [ initial ] algorithm { algorithm ";" | annotation ";" }

equation :
  ( simple_expression "=" expression
   | conditional_equation_e
   | for_clause_e
   | connect_clause
   | when_clause_e
   | IDENT function_call )
  comment

algorithm :
  ( component_reference ( ":=" expression | function_call )
```

```
    | "(" expression_list ")" ":=" component_reference function_call
    | conditional_equation_a
    | for_clause_a
    | while_clause
    | when_clause_a )
  comment

conditional_equation_e :
  if expression then
    { equation ";" }
  { elseif expression then
    { equation ";" }
  }
  [ else
    { equation ";" }
  ]
  end if

conditional_equation_a :
  if expression then
    { algorithm ";" }
  { elseif expression then
    { algorithm ";" }
  }
  [ else
    { algorithm ";" }
  ]
  end if

for_clause_e :
  for for_indices loop
    { equation ";" }
end for

for_clause_a :
  for for_indices loop
    { algorithm ";" }
  end for

for_indices :
  for_index {"," for_index}

for_index:
  IDENT [ in expression ]

while_clause :
  while expression loop
    { algorithm ";" }
```

```
  end while

when_clause_e :
  when expression then
    { equation ";" }
  { elsewhen expression then
    { equation ";" } }
  end when

when_clause_a :
  when expression then
    { algorithm ";" }
  { elsewhen expression then
    { algorithm ";" } }
  end when

connect_clause :
  connect "(" connector_ref "," connector_ref ")"

connector_ref :
  IDENT [ array_subscripts ] [ "." IDENT [ array_subscripts ] ]
```

A.7 Expression

```
expression :
  simple_expression
 | if expression then expression { elseif expression then expression } else
 expression

simple_expression :
  logical_expression [ ":" logical_expression [ ":" logical_expression ] ]

logical_expression :
  logical_term { or logical_term }

logical_term :
  logical_factor { and logical_factor }

logical_factor :
  [ not ] relation

relation :
  arithmetic_expression [ rel_op arithmetic_expression ]

rel_op :
  "<" | "<=" | ">" | ">=" | "==" | "<>"

arithmetic_expression :
```

```
  [ add_op ] term { add_op term }

add_op :
  "+" | "-"

term :
  factor { mul_op factor }

mul_op :
  "*" | "/"

factor :
  primary [ "^" primary ]

primary :
  UNSIGNED_NUMBER
  | STRING
  | false
  | true
  | component_reference [ function_call ]
  | "(" expression_list ")"
  | "[" expression_list { ";" expression_list } "]"
  | "{" function_arguments "}"
  | end

name :
  IDENT [ "." name ]

component_reference :
  IDENT [ array_subscripts ] [ "." component_reference ]

function_call :
  "(" [ function_arguments ] ")"

function_arguments :
  expression [ "," function_arguments | for for_indices ]
  | named_arguments

named_arguments:
  named_argument [ "," named_arguments ]

named_argument:
  IDENT "=" expression

expression_list :
  expression { "," expression }

array_subscripts :
```

```
    "[" subscript { "," subscript } "]"

  subscript :
    ":" | expression

  comment :
    string_comment [ annotation ]

  string_comment :
    [ STRING { "+" STRING } ]

  annotation :
    annotation class_modification
```