View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Lund University Publications - Student Papers

A CasADi Based Toolchain For JModelica.org

Björn Lennernäs

LUND UNIVERSITY

MASTER THESIS in Automatic Control 2013

Supervisors:

Toivo Henningsson - Modelon AB
Fredrik Magnusson - Department of Automatic Control, Lund University
Johan Åkesson - Modelon AB/Department of Automatic Control, Lund University

Department of Automatic Control
Lund University
Box 118
SE-221 00 LUND
Sweden

ISSN 0280-5316
ISRN LUTFD2/TFRT--5919--SE

© 2013 by Björn Lennernäs. All rights reserved.
Printed in Sweden by Media-Tryck.
Lund 2013

Abstract

Computer-aided modeling for simulation, optimization and analysis is increasingly used for product development in industry today, resulting in high demands on the tools used. A tool chain for transferring interpreted code of the modeling languages Modelica and Optimica from the simulation and optimization tool JModelica.org to CasADi has been implemented. CasADi provides several desirable features, most importantly an integrated and efficient automatic differentiation engine and the ability to interactively work with the systems expressed using it. The biggest problems solved to enable this were the creation of a representation of the mathematical systems described by Modelica and Optimica code that is integrated with CasADi, and the construction of a transfer scheme for moving information from the Java-based JModelica.org compiler to C++ in which CasADi resides. This was successfully achieved for a continuous subset of Modelica and Optimica that may contain functions.

Acknowledgements

I thank my advisors Toivo Henningsson, Fredrik Magnusson and Johan Åkesson.

- Toivo has provided mathematical and programming expertise, and has provided a lot of very useful feedback on wide variety of topics throughout the whole thesis.
- Fredrik has provided mathematical expertise and has also been the key person for providing the users perspective, as he has worked with CasADi before and aims to use the developed system.
- Johan has provided deep knowledge and expertise and has helped make important design decisions.

I thank Gustaf Söderlind and the staff at the department of numerical analysis at Lund University. They provided much help and encouragement for me, as a computer engineering student, when I wanted to learn numerical analysis, without which this thesis would not have been possible.

I thank Joel Andersson, one of the developers of CasADi, for providing quick and extensive help to questions and reflections posed.

I thank my fellow thesis colleagues at Modelon and the staff there, for friendly banter and miscellaneous help.

Lastly, I thank Erika.

Contents

1 Introduction **1**
1.1 Modelica . . . 1
1.2 Thesis goal . . . 1
1.2.1 Current state . . . 2
1.2.2 Goal . . . 2
1.3 Outline . . . 2

2 Background **4**
2.1 Mathematical formulation . . . 4
2.1.1 Differential equations . . . 4
2.1.2 Dynamic optimization problems . . . 5
2.2 Modelica and Optimica . . . 6
2.2.1 Declarative . . . 6
2.2.2 Syntax and semantics . . . 6
2.2.3 Translation process . . . 9
2.3 JModelica.org . . . 10
2.3.1 Compiler . . . 11
2.4 CasADi . . . 12
2.4.1 Automatic Differentiation . . . 13
2.4.2 Syntax and semantics . . . 14
2.5 Usage of the flat model . . . 15
2.5.1 Code creation . . . 15
2.5.2 XML . . . 17

3 Goals and motivation **18**
3.1 General goals . . . 19
3.2 Specific goals . . . 19

4 Implementation **20**
4.1 Implementation overview . . . 20
4.2 C++ model . . . 21
4.2.1 Model class . . . 22
4.2.2 Equations . . . 25
4.2.3 Functions . . . 26

4.2.4 Variables . . . 26
4.2.5 Optimization . . . 29
4.3 Transfer design . . . 31
4.3.1 Connection between Java and C++ . . . 31
4.3.2 Extending JModelica.org compiler . . . 32
4.3.3 Filling model . . . 36
4.4 Testing . . . 38

5 Benchmarks **39**
5.1 Solution comparison . . . 39
5.2 Timing . . . 41
5.3 Printing . . . 42
5.4 Stability . . . 43

6 Discussion **45**
6.1 Goal evaluation . . . 45
6.1.1 Specific goals . . . 45
6.1.2 General goals . . . 46
6.2 Future work . . . 47
6.3 Other considerations . . . 47

Chapter 1

Introduction

Computer-aided modeling of large and complex systems with the purpose of simulation, optimization and analysis is increasingly used in industry today. This puts high demands on the software tools and languages used to handle and express the models. The tools may need to handle more than 100 000 variables and equations, and the languages need to enable a clear and convenient modeling environment for the engineer.

1.1 Modelica

Modelica is a non-proprietary and domain-neutral modeling language that has gained popularity in the industry since its introduction in 1997. Users include automotive companies such as Audi, BMW and Ford, companies involved with power plants such as Siemens and ABB, and many more [1].

Modelica is an equation based, object-oriented and declarative language that lets the engineer work at a conveniently high level. In doing so, several responsibilities are given to the tools that handle the models. These often need to perform several manipulations on the models, e.g. *index reduction*, solving the initialization system and determining the order of execution for the equations, in order to obtain objects on which mathematical algorithms may operate. Examples of Modelica tools are AMESim, Dymola and JModelica.org [2].

1.2 Thesis goal

This thesis aims to implement a tool chain that connects the JModelica.org environment with CasADi. JModelica.org is an open source platform providing, among other things, a Modelica compiler and a computation environment. It also provides the Modelica extension Optimica, which allows for the formulation of Modelica based dynamic optimization problems [3].

CasADi is a minimalistic, general purpose, computer algebra system designed for nonlinear optimization. CasADi provides integrated support for

automatic differentiation (AD) and high-level interfaces to state-of-the-art solvers. CasADi is fast and efficient and comes with interfaces to Python and Octave, offering interactivity and convenience [4].

1.2.1 Current state

Currently JModelica.org can interpret Modelica and Optimica code and:

- Create code units that contains methods and other information that mathematical algorithms can use to simulate/solve them.
- Create XML-files that represents Optimica optimization problems. These files can be imported by other tools that can solve them.

The two different approaches offer different advantages and drawbacks. The code units have good coverage of the Modelica language but can be computationally slow, and once they are created they can not be changed. The XML format on the other hand offers incomplete coverage of the Modelica language (no hybrid systems), but the files retain their mathematical structure in such a way that they can be altered, and the tools that imports them may offer other advantages. CasADi offers such an import today, which can not represent Modelica functions but comes with the interactiveness, precision and speed that CasADi provides.

1.2.2 Goal

The goal is to lay the foundation for software that has the advantages of the XML-based transfer from JModelica.org to CasADi, as outlined above, but not the disadvantages. The performance of the implementation is important because it should be able to handle very large models. It should also be extendable so that more coverage of Modelica and Optimica (than this thesis provides), and other functionality, can be easily added.

These goals will be achieved by creating general representations of Modelica models and Optimica optimization problems that are integrated with CasADi, and by creating tools to populate them with data from the JModelica.org compiler. A large part of the difficulty of this task stems from the fact that the JModelica.org compiler is written in Java while CasADi is written in C++.

1.3 Outline

The report is organized as follows: Presented first, in Chapter 2, is a description of the mathematical formalism used, followed by a description of the tools involved, including how they can be used currently. Then the goals and motivations for this thesis are presented in Chapter 3. Thereafter the implementation is presented in Chapter 4: described first is the development of the CasADi integrated model representation, followed by the implementation of the information

transfer from JModelica.org to the model representation, then the description is concluded by a presentation of developed test methods. Thereafter some benchmarks that measure the correctness and performance of the implementation are presented in chapter 5. Presented last, in Chapter 6, is a discussion over how well the implementation realizes the goals as well as future perspectives.

Chapter 2

Background

An overview of the mathematical formulation that forms the basis for the languages Modelica and the extension Optimica is provided below. Then the languages Modelica and Optimica are described, followed by a description of JModelica.org and CasADi. The chapter is concluded by a description of how JModelica.org can use the interpreted models today.

2.1 Mathematical formulation

2.1.1 Differential equations

A common mathematical formulation used to describe the behavior of dynamic systems is *ordinary differential equations* (ODEs), i.e. equations on the form:

$$\dot{x} = f(x, t)$$

The use of ODEs in modeling has been popular for several decades, using e.g. block modeling with Matlab/Simulink. There exists a rich theory describing how different types of ODEs can be numerically solved as well as robust solvers.

But using ODEs for modeling is limiting. Commonly modeling of physical systems, and connections between them, are hard to express using only ODEs, e.g. physical modeling may give rise to algebraic loops that can not be expressed using ODEs. Even if the models can be expressed with ODEs, much work may be required by the modeler if the models are changed, e.g. by adding or removing components, resulting in a tedious modeling process. *Differential algebraic equations* (DAEs) do not require such extra work and they can describe algebraic loops, and they are therefore more suitable as a mathematical basis for convenient and powerful modeling. As such the mathematical foundation of Modelica models is DAEs, making it possible to use many powerful modeling concepts such as object orientation and connections between models [5].

More precisely Modelica allows the formulation of hybrid DAEs, which means that they may contain discrete parts. However this thesis restricts the under-

lying mathematical model to non-hybrid DAE systems, which are generally described by:

$$f(t, \dot{x}(t), x(t), w(t)) = 0$$

Where w denotes algebraic variables. This form is called the fully implicit form, and it is the one used throughout this thesis.

An important concept for DAEs is that of index, which relates to the number of times the equations in the DAE have to be differentiated in order to solve for the differentiated variables. By differentiating the DAE this way an ODE can be obtained, enabling the use of ODE solvers. Note however that numerical solutions of DAEs usually involve other techniques.

2.1.2 Dynamic optimization problems

The Modelica extension Optimica, that is provided by JModelica.org, allows for the formulation of *dynamic optimization problems* (DOP) based on Modelica models. DOPs represent a large class of problems, in this thesis however the DOPs considered are real valued *optimal control problems* (OCP). These problems consist of the dynamic system from the underlying Modelica model, some cost function that is to be minimized, the start and final time over which the problem is defined, constraints on the variables that make up the problem and some free variable. This may be expressed as:
Minimize the cost function:

$$f(x(t), w(t), u(t), p) \quad t \in [t_0, t_f]$$

Subject to:

$$\begin{aligned} f(t, \dot{x}(t), x(t), w(t)) &= 0 \quad t \in [t_0, t_f] \\ c_{\text{ineq}}(x(t), w(t), u(t), p) &\leq 0 \quad t \in [t_0, t_f] \\ c_{\text{eq}}(x(t), w(t), u(t), p) &= 0 \quad t \in [t_0, t_f] \end{aligned}$$

where u is the free control variable, c_{ineq} and c_{eq} are the inequality and equality constraints respectively, and p is the parameters.

The cost functions considered are of two different forms, and they are called the Lagrange and Mayer term. The Mayer form is expressed as a function at the final time point:

$$f_{\text{Mayer}} = M(x(t_f), w(t_f), u(t_f), p), \quad M \in C^2$$

The Lagrange form consists of a function that is integrated over the whole interval:

$$f_{\text{Lagrange}} = \int_{t_0}^{t_f} L(x(t), w(t), u(t), p), \quad L \in C^2$$

where C^2 denotes twice continuously differentiable functions.

2.2 Modelica and Optimica

Modelica is an object-oriented declarative language designed for convenient modeling of the dynamic behavior (i.e. over time) of complex heterogeneous systems. It is an equation-based language, and models are described using continuous DAEs and discrete equations.

The design of the language began in 1996 and it has been used in industry since 2000. It is provided for free and developed and maintained by the non-profit Modelica Association, which also freely provides the Modelica Standard Library containing generic model components from different domains. Motivating the effort was the need to construct a language that was not limited to a specific commercial tool, as was the case with most tools at the time, and that was domain neutral [6].

2.2.1 Declarative

Modelica models primarily consist of equations as opposed to assignments that are typically found in conventional imperative programming languages. This has important implications since these equations describe equality and it is generally not possible to execute the equations in the order that they are listed (no pre-defined causality). Instead the compiler will have to figure out in which order the equations should be exercised and what variables are set using them. Consider the following equation, which may occur in a Modelica model:

$$x + \sin(y) = z \cdot a$$

There are several variables on each side of the equation and it is not given how this equation should be used in a model. Depending on the manipulation in the compiler, the equation above could e.g. be used to calculate either one of x, y, z and a [7] [2].

2.2.2 Syntax and semantics

The basic Modelica program is a model, akin to a class in an object oriented programming language. The blueprint for models is very straightforward, and in its simplest form there is a list of variable declarations followed by an equation section. For example, the simple differential equation $\dot{x} = \lambda \cdot x$ can be modeled as:

```
model MyModelName
    parameter Real lambda = -1.0;
    Real x (start = 5.0);
  equation
    der(x) = lambda * x;
end MyModelName;
```

Below follows a brief walkthrough of the Modelica language, and what Optimica allows for on top of it.

Variables

A variable in Modelica may be given a wide variety of properties. They may have attributes such as start values, they can represent different types, and they can occur in different ways in the model (e.g. as inputs or as constants).

There are four primitive variable types; `Real, Integer, Boolean` and `String`. These are allowed to vary in different ways, which is called their variability. For example, they may be continuous like a differential variable, they may be discrete and only allowed to change values in certain situations, or they may be declared as `constant` or `parameter` and not be allowed to vary at all over time. Furthermore the variables may be declared as `input`, `output` or neither, which is called their causality.

A wide variety of attributes may be specified for a variable. Important examples are `min, max, nominal, start` and `unit`, or they may simply be given a value. There are a couple of important observations: some attributes may be declared as expressions, e.g `max = sin(p)`. This gives the second important observation: variables may have values that depend on other variables. A notable example of this is parameters whose values are given by an expressions of other parameters.

Other variable constructions that may occur in a model are instances of other classes and the structured forms arrays and records, where record is akin to a struct in C++ that may hold fields of arbitrary types. Custom variable types may also be built from the primitive types and these typically have a certain name and attributes, such as `unit`, preset.

Functions

An important part of Modelica is functions, and they follow a syntax similar to that of a model. A function definition consists of a variable field part and an algorithm section. The variable field part declares the inputs and outputs of the model as well as any internal variables. The algorithm section is somewhat similar to the equation section of a model, but an important difference is that it consists of assignments (see Section 2.2.1). A simple example, with functionality equivalent to the model above is:

```
model MyModelName
    parameter Real lambda = -1.0;
    Real x (start = 5.0);
    function f
        input Real in1;
        input Real in2;
        output Real out1;
    algorithm
        out1 := in1*in2;
    end f;
  equation
    der(x) = f(lambda, x);
```

```
end MyModelName;
```

The allowed syntax is very expressive:

- Regular variables, array and records may be used.
- Flow control is allowed.
- Other functions may be called.
- Variables that have been assigned may be reassigned.

A special class of functions consists of those functions that have several outputs. These are only allowed to occur in assignments, in contrast to normal functions that may occur in equations. An example of such a function is:

```
function f
    input Real in1;
    input Real in2;
    output Real out1;
    output Real out2;
algorithm
    out1 := in1*0.5;
    out2 := in2+out1;
end f;
```

Which can occur in the equation section as:

```
 (a,b) = f(x,y)
```

Other features

In addition to the elementary expressions and functions Modelica supports various types of control flow, including if-then-else statements, for statements and when statements.

Other important aspects of Modelica are the support for inheritance, packaging and connections between different models. These will not be described in detail here, as they are not visible after the compiler has interpreted and flattened the model (see Section 2.2.3).

Optimica

The Optimica extension provides a few new concepts that are needed to represent DOPs. Optimica is provided on top of Modelica; the dynamic system provided by Modelica is one of the conditions in the DOP (see Section 2.1.2). To reflect this, Optimica code that describes a DOP is called a class, as compared to a model for Modelica.

To create an Optimica class the keyword `model` is replaced by `optimization`. At the top the `objective` or `objectiveIntegrand`, corresponding to the Mayer

and Lagrange term respectively, are defined. The start and final times for the problem are also defined.

Optimica allows constraint to be placed on variables. There are two kinds of constraint, path and point constraints. Path constraints have to be fulfilled during the entire simulation, whereas point constraints have to be fulfilled at certain time points. These are declared in a new section, called constraint.

Optimica defines new unit attributes: `free` and `initialGuess`. The `free` flag determines whether the variable is free in the optimization, and `initialGuess` is used to provide an initial guess for a variables value, which may sometimes enhance solver performance.

An optimal control problem based on the *Van der Pol* (VDP) oscillator, with a Lagrange term as the cost function and constraints placed on the control signal `u`, can be represented as:

```
optimization vdp (objectiveIntegrand = cost,
                      startTime = 0,
                      finalTime = 1)
  // The states
  Real x1(start=0,fixed=true);
  Real x2(start=1,fixed=true);

  // The control signal
  input Real u;

  Real cost(start=0,fixed=true);
equation
  der(x1) = (1 - x2^2) * x1 - x2 + u;
  der(x2) = x1;
  der(cost) = x1^2 + x2^2 + u^2;
constraint
   u<=0.75;
end vdp;
```

The keyword `fixed` used above is not the opposite of `free`. It is a Modelica keyword and it determines how the value is handled during the solution of the initialization system (see Section 2.2.3 below). The `free` keyword is assumed to be true for the `input` variable and false for the other variables.

2.2.3 Translation process

It is the responsibility of the compiler to interpret and error check the Modelica code, and to transform it into something that can be used for computation. An important part of this process is called flattening, which transforms the code into a set of variable declarations and equations (replacing inheritance and component structures with appropriate variables and equations). The result of this process is called a flat model and it is generally manipulated further,

in order to make it possible to use with solvers. Examples of manipulations include index reduction (since many solvers require that the system is on ODE-form), *block lower triangular* (BLT) transformation (to sort the equations and make it more efficient to use with solvers) and *tearing* (to further enhance the performance).

The output from this process is a hybrid DAE, representing the mathematical structure of the original Modelica code. The next step is to connect this information with solvers; generally this is done by compiling it into c-code which is linked to solvers.

To be able to simulate the system the variables must be assigned start values. This is the responsibility of the initialization phase. This process generally gives rise to a set of initial equations, which are used to assign values to the variables [8].

2.3 JModelica.org

JModelica.org is an open source platform for simulation, optimization and analysis of complex Modelica based systems. JModelica.org allows for translation of Modelica and Optimica code and it provides a simulation and optimization environment. JModelica.org grew out of research at Lund University, and is now maintained and developed by Modelon AB in collaboration with Lund University.

Overview

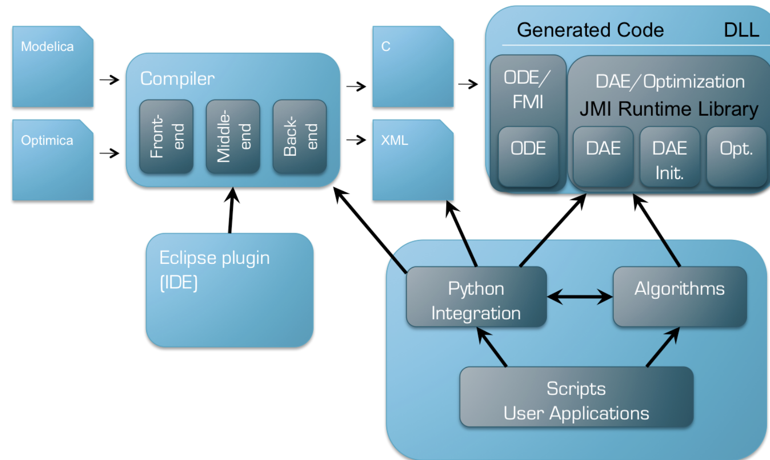


Figure 2.1: Overview of the JModelica.org environment.

An overview of the JModelica.org platform is shown in Figure 2.1. For the user of JModelica.org the integration with Python is very handy, offering convenient

ways to interact with the models, solvers and their result.

The compiler consists of three different parts: the front-end, middle-end and back-end. The front end is where the functionality for parsing and error checking models is located, and it is where the model is first flattened. The output from the initial flattening is used in the middle end where all manipulations, e.g index reduction and BLT transformation, are performed. The final flat model, that is the output from the middle end, is then used in the back-end. For a discussion of how the final flat models may be used the reader is referred to Section 2.2.5.

2.3.1 Compiler

JastAdd

The JModelica.org compiler is implemented using JastAdd, which is a tool designed for modular and extensible compiler construction. JastAdd uses an abstract grammar specification for describing and generating a Java class hierarchy, represented as an *abstract syntax tree* (AST). An AST is a common way to represent the data created by compilers, and the nodes in the AST typically maps to an element in the language [9] [10].

An example of an AST is shown in Figure 2.2, representing the expression $1 + 2$.

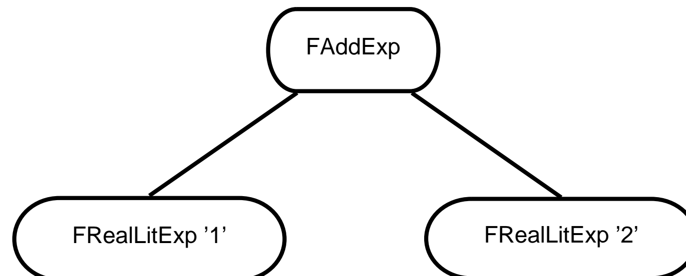


Figure 2.2: An AST for the expression $1 + 2$.

The specification that produces some of the Java classes needed for the above AST may look like:

```
FBinExp : FExp ::= Left:FExp Right:FExp;
FAddExp : FBinExp;
FLitExp : FExp;
FRealLitExp : FLitExp ::= <Value:double>;
```

Working through the statements:

- `FBinExp : FExp ::= Left:FExp Right:FExp;` Creates the class `FBinExp` that is a subclass of `FExp`, and that has two expression as children (called `Left` and `Right`).

- `FAddExp : FBinExp;` Creates the class `FAddExp` that is a subclass of `FBinExp`
- `FLitExp : FExp;` Creates the class `FLitExp` that is a subclass of `FExp`
- `FRealLitExp : FLitExp ::= <Value:double>;` Creates the class `FRealLitExp` that is a subclass of `FLitExp`, and that holds a `double` called `value`

JastAdd also provides an easily extendable and modular way to fill the produced classes with functionality, using aspects. The syntax is rather straightforward, and the developer may declare aspects and in them define behavior of classes that are related to the aspect. Continuing with the above example, a method that returns the value of the `FAddExp` might be implemented as follows:

```
aspect Arithmetic {
    public double FAddExp.calculate() {
        return Left.value() + Right.value();
    }
    public double FRealLitExp.value() {
        return value;
    }
}
```

This will add the `calculate` method to `FAddExp`, and the `value` method to `FRealLitExp`. Note that this is a simplified example and it would need more methods to make the resulting Java classes compile.

The observant reader might wonder why the methods and classes in the examples above are prefixed with an "F". This is because they are inspired by the Java classes that make up the AST for the **F**lat model in the JModelica.org compiler.

Flat model class

There are actually two different compilers in JModelica.org, one for handling Modelica models and the other for handling Optimica models. The flat model is represented by the classes `FClass` and `FOptClass` respectively. These contain the model, and they keep such information as variables, equations and function declarations.

2.4 CasADi

CasADi is an open source minimalistic computer algebra system that offers eight different flavors of automatic differentiation (AD). Written in C++, but offering full featured front ends to Python and Octave, it is designed to be a low level, fast and efficient tool for the developer and user of algorithms for numerical optimization. It offers back ends to state of the of art solvers such as Sundials, IPOPT, WORHP and KNITRO [11] [12].

2.4.1 Automatic Differentiation

In simulation and optimization there often arises a need to use iterative methods such as Newton's method to handle non-linear problems. These methods often require knowledge of the derivatives of the problems expressions; hence it is of great importance to efficiently and accurately construct them. The classical way of constructing them is either through numeric or symbolic differentiation, i.e. constructing a full expression for the analytical derivative. These methods have significant drawbacks though. Numerical differentiation is often inaccurate and symbolic differentiation is slow and faces many difficulties when dealing with large expressions. Automatic differentiation solves those problems.

By taking advantage of the fact that expressions consist of a sequence of elementary operations and functions (e.g plus/minus and sin/cos), and by utilizing the chain rule, derivatives can be constructed fast and with the same accuracy as symbolic derivatives. There are different approaches, depending on e.g the order in which the expression graphs are traversed. In Figure 2.3 the use of forward AD to find the derivative of the function $f(x_1, x_2) = x_1 x_2 + \sin(x_1)$ is illustrated.

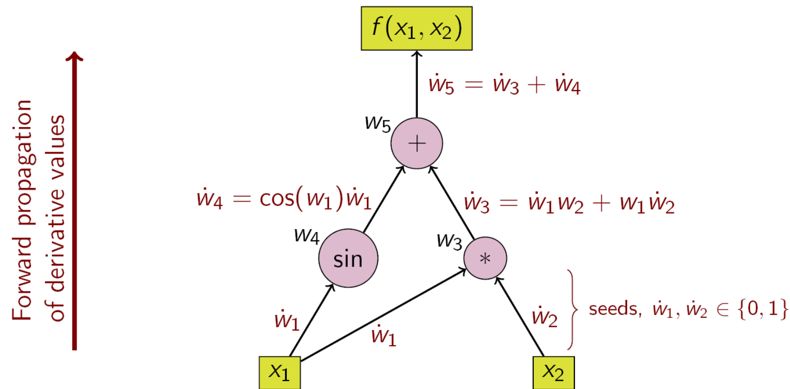


Figure 2.3: Illustration of forward AD.

To find the derivative with regards to a variable, e.g x_2, the derivative of the AD variable corresponding to it, $\dot{w}_2$, is set to one and the variable corresponding to the derivative of x_1 to zero. These seeds are then used to calculate the value of the AD variable corresponding to the derivative of the full function, $\dot{w}_5$. In tabular form:

Node	Derivative expression	Resulting variable value
$w_1 = x_1$	$\dot{w}_1$	0 (seed)
$w_2 = x_2$	$\dot{w}_2$	1 (seed)
$w_3 = x_1 \cdot x_2$	$w_1 \cdot \dot{w}_2 + \dot{w}_1 \cdot w_2$	w_1
$w_4 = \sin(x_1)$	$\cos(w_1) \cdot \dot{w}_1$	0
$w_5 = x_1 \cdot x_2 + \sin(x_1)$	$\dot{w}_3 + \dot{w}_4$	w_1

Thus the derivative of f with regards to x_2 is given by the variable w_1, corrsponding to x_1 [13].

2.4.2 Syntax and semantics

CasADi is a symbolic framework that lets the user define and use expressions and variables in an intuitive way. The syntax, and inner workings, of CasADi centers around the two types `SX` and `MX` and their dependent classes, and the `FX` class for handling functions. Consider a simple example, how to construct the expression $a \cdot b$. In the Python terminal this may be achieved by writing:

```
a = SX("a")
b = SX("b")
exp = a*b
print exp
=> (a*b)
```

There are several built in operations that can be used, e.g. the jacobian of the expression above with regards to the variable `a` may be found using:

```
jacobian(exp, a)
=> Matrix<SX>(b)
```

Here the type `SXMatrix` is shown, which is just a matrix with elements of type `SX`. The variables that occur in an expression may be substituted for others:

```
sub = substitute(exp, a, SX("new"))
print sub
=> Matrix<SX>((new*b))
```

Functions can also be easily constructed, e.g the function $f(a, b) = a \cdot b$:

```
f = SXFunction([a,b], [exp]) # Make an SXFunction instance
f.init() # Initialize, f can now be called
f.input(0).set(1) # set a = 1
f.input(1).set(2) # set b = 2
f.evalute()
print f.output()
=> 2
```

An important difference between `SX` and `MX` lie in what types of expressions that they allow. The `SX` type is used to build up scalar unary or binary expression ($\mathbb{R} \rightarrow \mathbb{R}$ or $\mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$), while `MX` expressions are allowed to be "general multiple sparse-matrix valued input, multiple sparse-matrix valued output functions: $\mathbb{R}^{n_1 \times m_1} \times \ldots \times \mathbb{R}^{n_N \times m_N} \rightarrow \mathbb{R}^{p_1 \times q_1} \times \ldots \times \mathbb{R}^{p_M \times q_N}$ " [14]. Importantly `MX` allows for building expressions where function calls are a part. However, `SX` is as a rule faster and more intuitive to work with.

For example, to create an `MX` expression where a function call is a part in the Python terminal:

```
## Construct function
funcVar = MX("fv") # Variable occurring in function equation
funcExp = funcVar**2 # Function expression
f = MXFunction([funcVar], [funcExp])
f.setOption("name", "myFunction") # Gives the function a name
f # prints function
=> function("myFunction")

## Construct call
f.init() # Need to initialize it before it is called
funcArg = MX("arg") # The call argument
call = f.call([funcArg])[0] # Make the call and extract MX
call # Prints it
=> MX(function("myFunction").call([arg]){0}) # The call is an MX object

## Construct expression with a call
var = MX("var")
eq = var * call # Construct the expression
eq # prints it
=> MX((var*function("myFunction").call([arg]){0}))
```

2.5 Usage of the flat model

This thesis concerns the use of the final flat model, after compilation and manipulation (see Section 2.2.3 and 2.3 for an overview of these concepts). The ways in which this model is used today in JModelica.org is described below. In short it is used either to create and compile code, which is linked with solvers, or too create an XML-file containing a representation of the model which can later be imported into other tools (e.g. CasADi).

2.5.1 Code creation

The standard way of dealing with Modelica and Optimica code in JModelica.org is through the generation of code-units, described below. These code-units offer very good coverage of Modelica and Optimica. The drawbacks are that it can take a long time to compile and then run the code, and importantly that it is a static process; the code units supports very little post compilation tinkering. Advantages are that the units can be saved for later and that they are not necessarily bound to the JModelica.org environment, and that both the solver and compiler do not have to be activated at the same time (thus reducing the burden on working memory).

FMI

JModelica.org implements the *Functional Mock-up Interface* (FMI) for generating code-units for Modelica models. FMI defines an open and standardized

interface for accessing the properties of a model, e.g. the variables and methods to evaluate them. Thus code-units that implement FMI that were generated by one tool may later be simulated using another FMI compliant tool. Another important aspect of FMI is the intention that units may be connected, allowing co-simulation and virtual product assembly.

Practically code-units that implement this interface are called *Functional Mock-up Units* (FMU). The FMU is a zip file (with .fmu file ending) and it contains an XML-file, mainly containing variable definitions, and C- and dll-files containing the rest of the information, such as functions to evaluate the model equations [15].

JMI

JModelica.org defines the *JModelica.org Model Interface* (JMI) for generating code-units, called *JModelica.org Model Unints* (JMU), for Modelica and Optimica. JMI handles optimization information and dynamic systems on a form similar to what is used in this thesis, therefore some internal aspects of JMI are described in more detail here.

JMI consists of mainly two parts, the DAE system of the model, including the initilization system, and the optimization information, containing e.g. constraints and cost functions. Accesses to these are offered as a collection of functions, where the input arguments are of a set of different variable kinds. These variable kinds consist of:

- Dependent and independent parameters for all primitive types. As explained in Section 2.2.2, parameters may be defined by expressions, and these expressions may depend on other variables, thus making the parameter defined in this way a dependent parameter.
- Dependent and independent constants for all primitive Modelica types.
- Real derivative $\dot{x}$ and state variables x.
- Real algebraic variables
- Real, Integer and Boolean inputs
- Time
- Discrete real variables
- Integer and Boolean variables

For more information see [16].

2.5.2 XML

There is an XML format for representing continuous time DAE based models from equation based modeling languages, including Modelica. JModelica.org allows for creation of XML-files based on this format, and CasADi has functionality to import these XML-files.

Advantages of this format are that, like for code creation, the files may be saved for later, that they are not necessarily bound to a specific tool, and that the tool that uses the XML-files does not need to be activated at the same time as the tool that creates them. Additional advantages are achieved since the XML-format saves the model on a form that may allow further manipulation. The disadvantages come from the limitations of the format itself, and the tools currently supporting it.

XML format

The XML framework was proposed in [17]. Important goals for the project was to provide a framework that was neutral with respect to model usage, easy to use, maintain and understand.

To achieve this, the format has a representation as close to a set of equations and variables as possible. Information about inheritance, complex data structures and so forth are therefore not part a of the format. The starting point was the XML-format provided by FMI (Section 2.5.1), which was extended with new modules for expressions, equations, functions and algorithms. An important limitation for the format is that it only handles continuous time models, i.e. no hybrid models are allowed.

The referenced paper also describes a test implementation in JModelica.org. This implementation supports the export of models via an extended version of this framework, for handling Optimica models. The authors noted that it was quite easy to construct the implementation since the AST in the JModelica.org compiler, see Section 2.3.1, is easily mapped to the format.

CasADi import and SymbolicOCP

CasADi offers a module for importing and representing models described on the format above called `SymbolicOCP`. The support for the format is not complete; importantly it does not support functions. The model representation is also fairly basic. Despite these limitations numerical algorithms using this functionality have been successfully implemented [18].

Chapter 3

Goals and motivation

As stated in the introduction, the goal of this thesis is to develop software to represent Modelica and Optimica models using CasADi, and to transfer models from the JModelica.org compiler to this new representation. The idea is to start building a new way for JModelica.org to handle compilation and handling of models, which is to be used extensively in the future. The future use might not only be limited to transferring the final flat model, but some parts of the manipulation capacities of JModelica.org, such as index reduction and BLT-transformation, may also be moved here (see Section 2.2.3 for a presentation of these concepts). Furthermore numerical algorithms will be developed to utilize the implementation.

Motivating this effort is the desire to use the many advantages offered by CasADi, such as symbolic representation, AD and interactivity. Listing these advantages:

- CasADi has an integrated and highly efficient AD-engine, which provides a great advantage for many simulation and optimization tasks.
- CasADi, and its solver interfaces, provide a speed increase over the traditional code based approach.
- CasADi provides a comprehensive environment for developing numerical algorithms.
- CasADi is interfaced with Python, which means that the interactivity provided by many Python prompts such as IPython, can be utilized together with CasADi. Thus the user may interactively work with CasADi data structures, manipulating them as they please.
- Overcome limitations of current schemes, as presented in Section 2.5.

Given these long term goals and motivations, and the need to comply with the limited scope of this thesis, a list with general design goals for the developed software as well as a list with specific functionality goals for this thesis, are presented below.

3.1 General goals

To enable the long term goals, it is important that care is taken when constructing the system. For the part of the system that consists of the model, the systems should be designed such that it is:

- Easy to understand, hopefully many will try to understand, work with and maintain the system.
- Extensible, so that algorithms may be developed for it, and so that support for current and future versions of Modelica and Optimica can be added.
- Efficient, extremely large models, with more than 100 000 variables and equations, should be possible to handle.

In short, the model should be as minimalistic as possible without locking out future additions to it. Similarly for the design of the transfer software, it is important that it is as efficient, general and extensible as possible.

3.2 Specific goals

The implementation should be able to transfer and represent a real and continuous subset of Modelica and Optimica that may include functions. Tests should also be implemented to make it possible to measure the correctness and performance of the implementation.

The biggest feature that will not be supported is hybrid systems. Other important Modelica construct that will not be supported are arrays and records, other variables types than real and handling of dependent parameters.

Chapter 4

Implementation

With the background and goals established, it is time to describe the implementation. An overview of the system is provided below, followed by a walkthrough of what this implies about the structure of the system. Thereafter that structure is described in more detail, with sections describing the C++ model, the transfer functionality and the test functionality.

4.1 Implementation overview

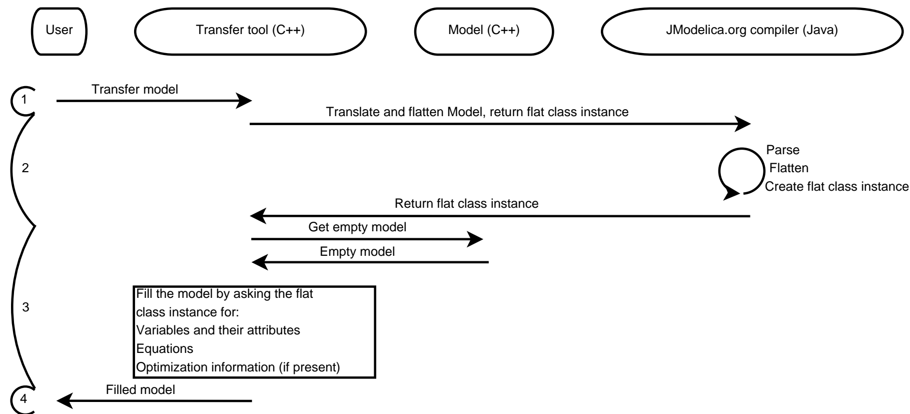


Figure 4.1: Overview of the transfer

Figure 4.1 presents an overview of the model transfer, four main parts are displayed:

1. The user decides what model to transfer. The transfer tool is written in C++, but it may be called from any language that may be interfaced with it (e.g. Python).

2. The transfer tool calls the JModelica.org compiler and asks it to return an instance of the flat class, representing the final flat model (see Section 2.2.3 and 2.3).

3. The transfer tool creates a new empty instance of the C++ model representation. The main work of the transfer tool then follows, i.e. to use the information in the flat class instance to fill the C++ model.

4. The filled C++ model is returned to the user, who may e.g. connect it to solvers or manipulate it.

The scheme above requires the construction of three main components. Firstly the C++ model must be constructed, which can represent Modelica and Optimica. Secondly a way to transfer the model from JModelica.org to C++ must be implemented. This task can also be dividided into subtasks:

- Enable the transfer of information between C++ and Java.
- Extend the JModelica.org compiler to make use of these transfer mechanics.
- Use the extended JModelica.org compiler and the transfer mechanics to fill an instance of the C++ model.

Lastly, tests should be implemented that solves problem represented by the transferred models.

Below the implementation of these different systems are described, in the same order as presented here.

4.2 C++ model

In Chapter 3 the goals for this thesis, and this model, were presented. The task is to create a model that is capable of representing the DAE system for Modelica models, that may include functions and the optimization information provided on top of that system by Optimica. Importantly, CasADi should be used to represent and manipulate mathematical expressions and functions.

Recalling the mathematical formulation described in Section 2.1, we know that the model should be able to keep initial and model equations, variables with differing properties and attributes (see Section 2.2) and optimization information such as constraints and Lagrange and Mayer terms. In order to have an understandable and extendable system, the choice was made to make it object oriented. Thus the natural skeleton of the system is to construct a main class, the Model class, which keeps all information about the DAE system. This means that it will have to keep variables, equations and functions which are naturally represented with their own classes. Lastly, since the optimization information is provided on top of the DAE system, it is natural that another package that keeps the optimization information is constructed, and that this system then keeps an instance of the model.

Having decided on a skeleton for the model the next important problem is how it should be integrated with CasADi. The elements that can be expressed are variables and the expressions in which these variables occur. Two important considerations here are:

1. The expressions must be made up of the same variables as are present in the model (compare this to the CasADi expression example presented in Section 2.4.2).
2. It must be decided on which CasADi expression type, `SX` or `MX`, that should be used.

For the first question, the choice is that this is not really the responsibility of the model, rather it is the responsibility of the transfer mechanics to make sure that the model it creates fulfills this. Thus the model should be constructed so that it can be filled in a manner that does not hinder the transfer mechanics in this regard. The answer for the second question is given by the fact, as presented in Section 2.4.2, expressions made using `MX` are much more general, and that they may contain function calls, thus `MX` will be used.

All the classes that make up the model also overrides the stream insertion operator `<<` for easy pretty printing.

4.2.1 Model class

The responsibility of the model class is to provide a way to store and access initial equations, model equations, functions and variables. It is through the model class that most of the interactions will occur, i.e. interactions from the user, the transfer mechanics, solvers etc., so it is important that care is taken to provide a sensible and efficient API.

The internal handling and the API for handling the variables and equations in the model class are described below. A simplified UML diagram (that does not show the complete class hierarchy for variables) is provided in Figure 4.2.

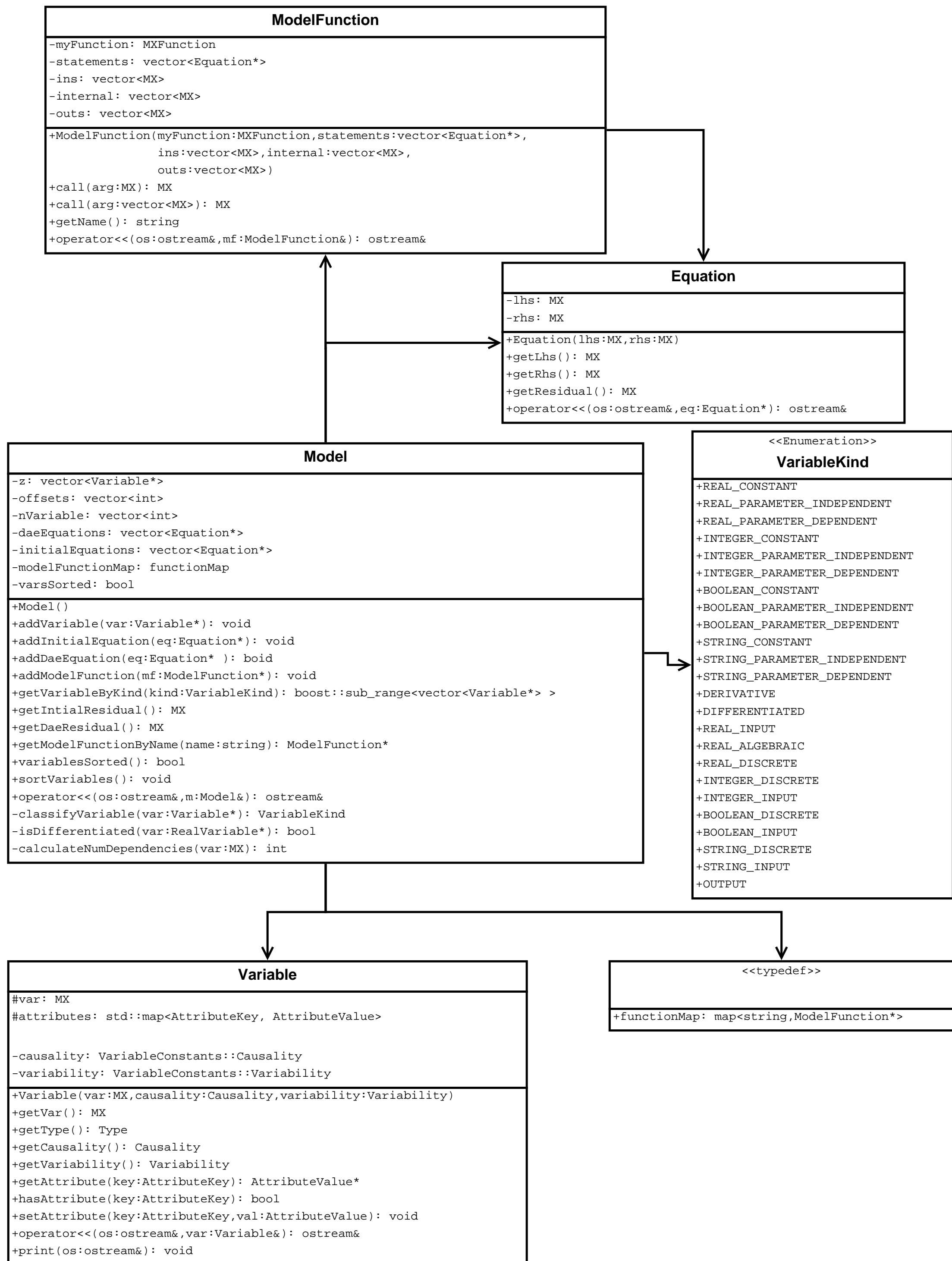


Figure 4.2: Simplified UML diagram showing model class and dependent classes. Does not show variable class hierarchy or optimization.

Variable classification

There are several different types of variables, e.g. derivative, state and algebraic variables but also parameters, constants and variables that occur as input and output. Naturally these variables need to be handled differently, and numerical algorithms often need to have the variables categorized to some extent. Thus it is natural that the model class provides these variables sorted into sensible categories.

In Section 2.5 a walkthrough of the current ways in which JModelica.org handles models were presented, and here the JMI specification provided a set of variable categories based on the variable attributes type, variability and causality introduced in Section 2.2.2. A similar approach will be used here. The provided variable categories are:

- Real, Integer, Boolean and String constants
- Real, Integer, Boolean and String parameters which are either dependent or independent
- Derivative
- Differentiated (state variables, that has derivative variable tied to them)
- Algebraic
- Real, Integer, Boolean and String inputs
- Real, Integer, Boolean and String discrete variables.
- Output

For internal and external use an enum with these categories is provided.

A variable will belong to only one category, which is uniquely determined by its type, causality and variability **except** for derivative, algebraic and differentiated variables. These all have the same type Real, the same variability Continuous and the same causality Internal. Thus the variable class needs to provide some information that can be used to discriminate among them.

Variable handling and API

The variables need to be stored in some way, they need to be classified according to the categories above and the model should provide methods to add and access variables.

Starting with storage, for efficiency reasons the variables are kept as pointers in a vector, in order to avoid them being copied as they are moved around. For simplicity the variables are kept in a single vector, making it easy to e.g. perform manipulations on all variables. This vector can then be sorted according to the variable categories. To make it easy to work with, e.g. to extract variables of certain categories, two additional vectors are provided: the first with offsets,

that points to where the first element of a certain variable category resides in the vector, and the second with the number of variables in each category.

As for the API, two main methods are offered: `addVariable`, that takes an instance of a `Variable` (see below), and `getVariableByKind`, that takes as an argument a value of the variable category enum. For efficiency reasons the decision to use lazy classification was made, which means that as variables are added to the model they are not categorized right away. Otherwise the whole variable vector might need to be manipulated every time a variable is added, creating a very unattractive time complexity for filling the model. To sort the vector a function `sortVariables` is added, along with the method `variablesSorted` to check whether the variable vector is currently sorted.

When asking for variables of a certain kind it would be inefficient to create a new list by probing the variable list. It would be preferable if a subrange of the list could be provided instead, i.e. some construct that would provide access only to certain elements in the list. The `BOOST` library provides such functionality as part of the range collection, see [19] for more details. When a variable category is asked for a special kind of iterator that can be used to access a subrange of the variable vector, containing the relevant variables, is returned. The advantage of using this approach is that it is much cheaper to construct this iterator than it is to construct a new vector and fill it with the desired variables.

Equation handling

The handling of equations is much simpler than that for variables, as there are basically only two kinds: initial and model equations. These are kept in two vectors, one for each kind, containing equation pointers.

Two methods for adding equations are provided, one for each kind. They can be retrieved on their residual form, i.e. for an equation with a left hand and right hand side the residual is the right hand side minus the left hand side. The residual form is provided as an `MX` object, containing all the residuals for the specified equation type.

Function handling

Functions are kept in a map in the model, with their name as the key used to access them. Two methods are provided, one to add a function, and one to retrieve a function using its name.

4.2.2 Equations

Equations are simple to represent. The model class keeps a list of model equations and initial equations; however both of these equations types are easily represented by the same equation class. This class should keep MX objects that represent the equation. A straightforward and useful approach is to keep an `MX` object each for the left and right hand side of the equation. Access methods for the left and right hand side and the residual are provided. The `MX` objects are

capable of representing functions with several outputs, as described in Section 2.2.2. See Figure 4.2 for an UML of the class.

4.2.3 Functions

A major problem for this thesis was how to represent functions and function calls that can occur in Modelica models. Not only does this need to work with the mathematical representation that is used in the thesis, but it should be efficient, simple and possible to extend in the future. Luckily it was found that the CasADi class `MXFunction` would probably suffice.

`MXFunction` is convenient and easy to use, and it will probably be possible to extend it in the future to account for discrete elements and flow control. To create an `MXFunction` a list of input variables and equations for the outputs are needed, one equation for every output variable. This object can be called, using other `MX` variables as arguments, yielding a new `MX` object that can be a part of other `MX` expressions.

As will be made clear below in the transfer section, all of the function calls that occur in the model's equations are transferred automatically as part of `MX` expressions constructed in the JModelica.org compiler. So there is no need to transfer the `MXFunction` objects to have a complete mathematical representation of the model equations.

However, in order to make it possible to construct new function calls using the `MXFunction` generated in the JModelica.org compiler, these need to be transferred and stored. As one of the goals is to provide interactivity for the user, this is a needed feature.

Functions should also be printable, to present the user with information about the statements and variables that occur in them. Recalling the structure of Modelica functions presented in Section 2.2.2, functions have a list of statements, a name and variables that occur as input, output or that are internal to the function.

To accommodate these requirements the class `ModelFunction` is introduced. This class has a name, corresponding to the name of the `MXFunction`, the `MXFunction` itself and the list of statements (represented by the `Equation` class) and variables. To create a new function call the class provides methods where `MX` objects are passed in as arguments, returning a new `MX` object that represents the function call. See Figure 4.2 for an UML of the class.

4.2.4 Variables

The variable class, or class hierarchy, should provide a way to represent the various types of properties and attributes that variables in Modelica might have. Furthermore it should keep a CasADi object that represents the variable. For this thesis, the goal is to provide coverage for real variables, however the design should make it easy to add support for integer, boolean and string variables.

Requirements and API

In Section 2.2.2., the variable properties type, variability and causality were presented. These properties are used by the Model class to categorize the variables, thus they should store these properties and they should provide methods for accessing them. However, it was noted above that these types are not sufficient to uniquely classify all variables into one of the provided categories, the problem is that state, derivative and algebraic variables share the same properties. Thus real variables need to provide additional information, which can be used to distinguish them.

Variables may also have attributes, and the variable class needs to provide methods to both set and retrieve them.

Solution

A main `Variable` class is introduced. This class provides access to methods for the three variable properties above, and keeps an `MX` object as the needed CasADi variable representation. The properties are encoded in three enums.

To represent the additional information needed for classification of real variables two additional classes, `RealVariable` and `DerivativeVariable`, are introduced. The `DerivativeVariable` class keeps a reference to its state variable. To determine whether a `RealVariable` is an algebraic or state variable the variables need to be checked against the references that the derivative variables keep. These classes also implement a method `isDerivative`, needed to find the derivatives in the first place.

The variable class keeps a map with strings as key and `MX` objects to handle attributes. This approach is simple, and attributes may be easily added and retrieved from the model. The approach offers the additional advantage that this variable class can be used for both Modelica and Optimica, as Optimica defines additional attributes over those found in Modelica (see Section 2.2.2).

An UML diagram for the resulting scheme is presented in Figure 4.3.

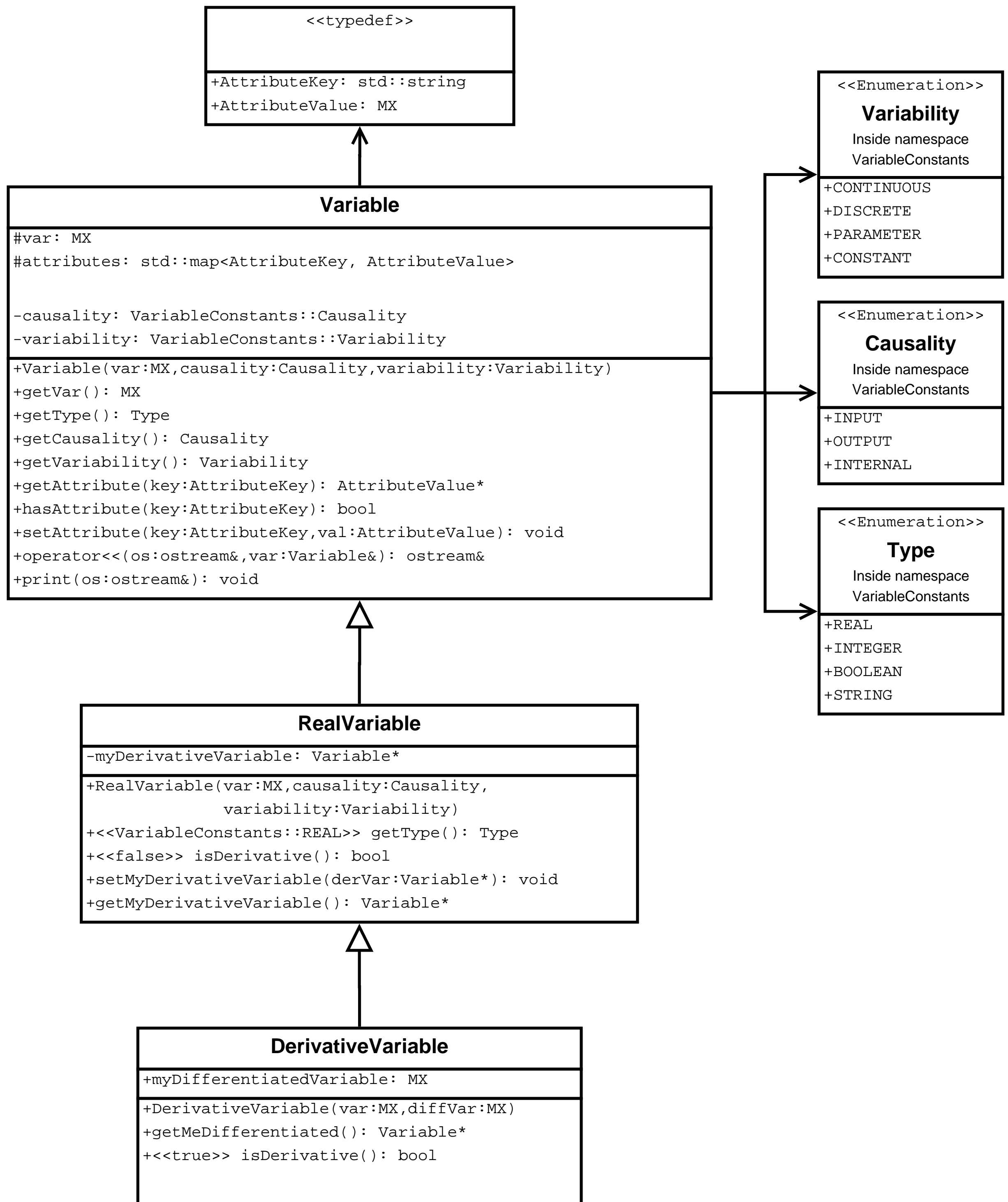


Figure 4.3: UML diagram for Variable class hierarchy.

4.2.5 Optimization

Optimica is an extension of Modelica, and adds optimization information on top of Modelica models. It is therefore natural to separate them; the idea is that the model is independent of the optimization problem, and all the optimization information should be independent of the model formulation. A new class `OptimizationProblem`, which holds a pointer to a model, is therefore introduced.

Optimica adds constraints, cost function, new variable attributes and start- and finaltime. All of these additions, except constraints, are simply represented as a single `MX` field in the `OptimizationProblem` class, with standard getters and setters. The constraints however are modeled with their own class. Optimica defines path and point constraints; however the decision was made to limit the representation to path constraints.

Constraints are very similar to equations, with a left hand and right hand side, typically a variable on the left hand side and some expression on the right hand side. There are three types of constraints, less than, greater than and equal to. The `Constraint` class thus holds a left and right hand side, and an enum value that describes which constraints type it is. The `OptimizationProblem` class keeps a list with all the constraints.

An UML diagram is presented for this part of the representation in Figure 4.4.

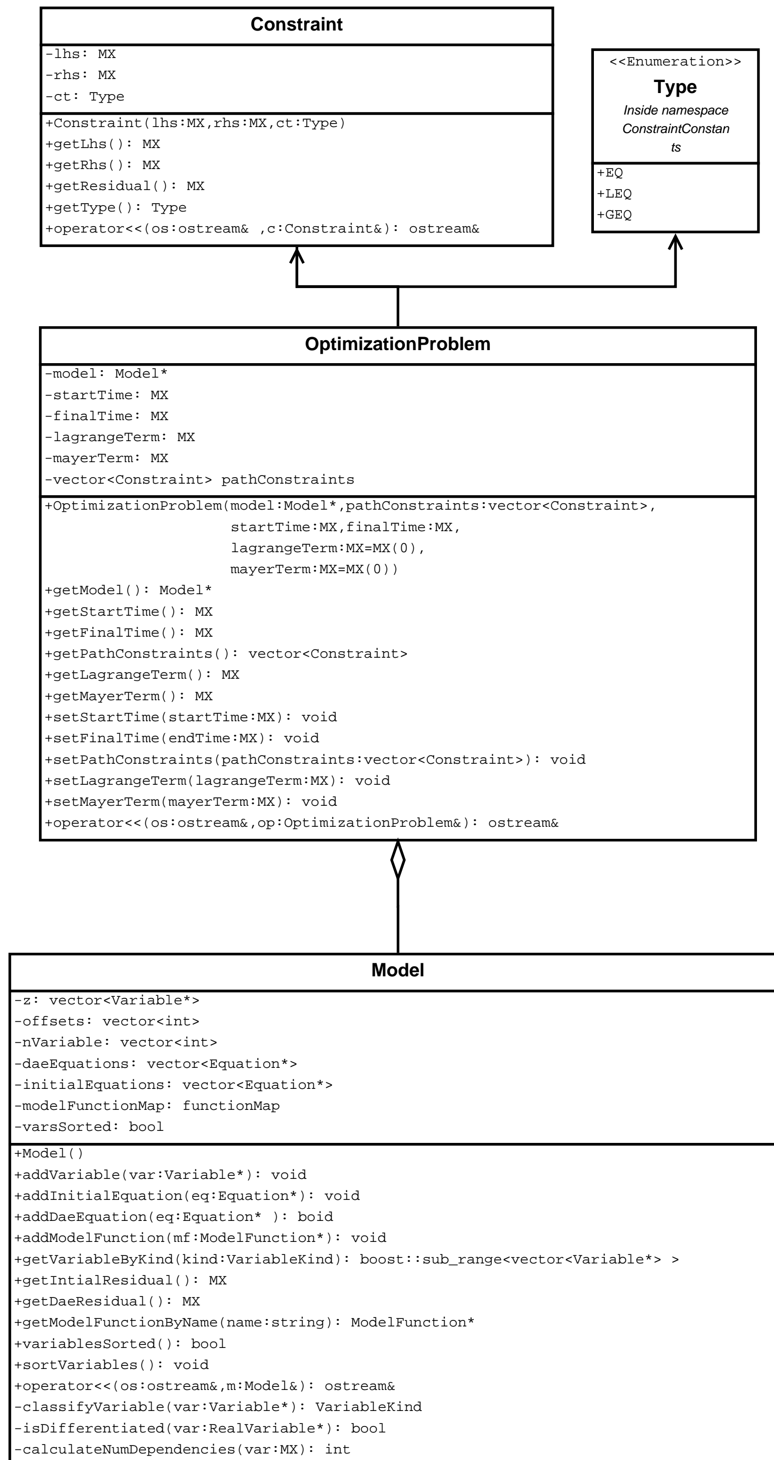


Figure 4.4: UML for Optimization extension

4.3 Transfer design

The second important task is to transfer the final flat model data from the JModelica.org compiler to the model described above. There are three major points that need to be addressed in order to accomplish this:

1. Building infrastructure for transferring information between Java and C++.
2. Extend the JModelica.org compiler to make use of these mechanics, so that it may be asked for data.
3. Build a transfer scheme, which calls on the JModelica.org compiler to obtain an instance of a flat class, representing the final flat model, which is then used to fill the model above.

Other considerations are, as described in Section 3.1, that we need it to be efficient and general.

4.3.1 Connection between Java and C++

Transfer scheme

The model representation resides in C++ along with CasADi, while the JModelica.org compiler resides in Java. The goal is to mirror the information represented by the JModelica.org compiler in an instance of the model representation. To do this, the AST (see Section 2.3.1) that represents the model in JModelica.org has to be traversed, and the information needed to create variables, functions, equations and so on has to be extracted. Two different approaches to extracting this information were considered:

The first approach is to traverse the AST and extract all information, such as names, values and attributes from Java to C++ and then use this information to create the CasADi objects needed for the model. The other approach is that JModelica.org itself is connected with CasADi, and that methods are implemented inside the JModelica.org compiler that creates the necessary CasADi objects. These CasADi objects can then be transferred to the model representation.

To determine which way was best investigations were performed by my advisor Toivo. Toivo determined that the best way to transfer information was by creating the CasADi objects directly inside the JModelica.org compiler. It was faster to call C++ from Java, and this constitutes a compromise where the fast transfer is used to create CasADi objects that can then be extracted in a convenient manner in C++. The transfer scheme is illustrated in Figure 4.5.

Transfer mechanics

The basis of the transfer is the utilization of the *Java Native Interface* (JNI) that allows calls between programs running in a *Java Virtual Machine* (JVM) and other applications in the operating system, e.g. C++ and C. For more

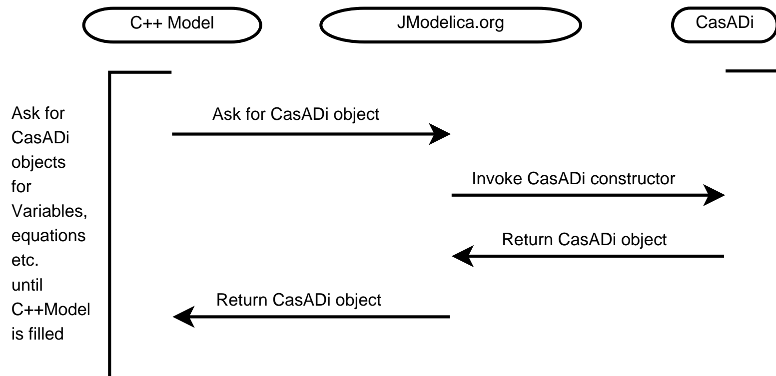


Figure 4.5: Overview of transfer scheme. C++ Model and CasADi resides in C++, JModelica.org resides in Java.

information see the Oracle JNI documentation [20]. To make it easier to use two other tools are utilized, which create easy to use interfaces that may be used in inter-language communications.

JCC is a tool that generates C++ code that wraps Java libraries, using JNI, in a way that resembles the original Java code. This allows Java to be called from C++. Similarly SWIG is a tool that can be used to generate Java code that wraps C++ libraries in the same way, allowing calls from Java to C++. For more information see the JCC homepage [21] and the SWIG homepage [22].

The foundation for this transfer was laid by my advisor Toivo. It was extended in several places as part of this thesis.

4.3.2 Extending JModelica.org compiler

As described above, the fastest way to do the transfer is to let the JModelica.org compiler construct CasADi objects, thus the goal is to create all CasADi objects here.

Aspects

As mentioned in Section 2.3, much of the functionality in the JModelica.org compiler is declared in aspects. JastAdd uses functionality described in different aspects to fill Java classes with methods and fields. The functionality aimed for here will not introduce new classes, only new methods and fields to existing classes. To this end, a new aspect is introduced, called `FExpToCasADi.jrag` (.jrag is the file ending used for aspects), wherein the methods needed in this thesis are declared.

The allowed syntax in aspects is quite expressive, however only a few concepts are needed here. The **`syn`** keyword is used to declare a new attribute for a class (synthesized attribute), e.g. the following declares that the class `A` should keep an attribute x and that its value is given by a certain Java expression:

```
syn T A.x() = Java-expr;
```

where T is the type of x.

If a function that is subclass of `A`, class `B`, wants to override the expression used to calculate the value x the following may be used:

```
eq B.x() = Java-expr;
```

Finally, if the intent is that the class stores the attribute, so that it is not recalculated each time it is accessed, the `lazy` attribute may be used:

```
syn lazy T A.x() = Java-expr;
```

Now `A` will store the attribute x as a field in the class, and it will only be computed once.

AST traversal

An AST that shows how the expression $a + b \cdot c$ is represented in JModelica.org is displayed in Figure 4.6. There are binary and unary expressions representing all the primitive operations and functions. The expressions keep references to their children. Thus expression graphs like these may be traversed by asking for the children of the nodes, until leafs are reached.

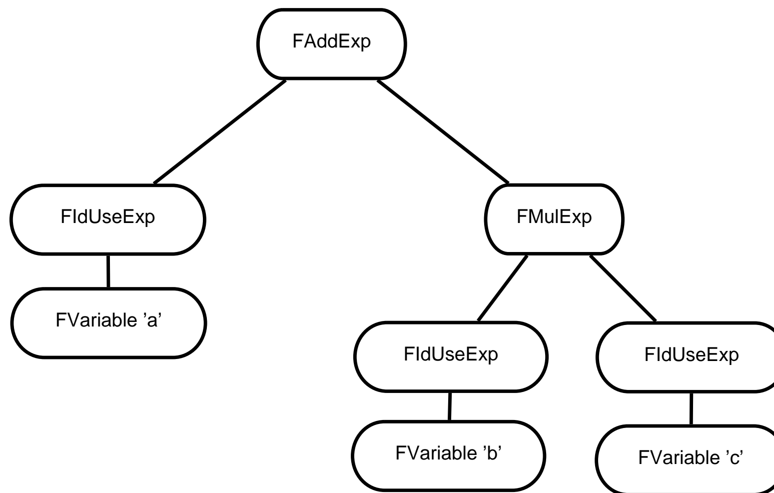


Figure 4.6: JModelica.org AST of the equation $a + b \cdot c$.

Variables

To create `MX` objects for variables a lazy synthesized attribute `MX_variable_CasADi`, with type `MX`, is introduced to the abstract variable class `FAbstractVariable`, from which the other variable classes inherit. The expression that determines

the value of this attribute is then overriden in the subclasses, e.g the expression that determines the MX value of the FVariable class is computed like:

```
eq FVariable.MX_variable_CasADi() = new MX(getFQName().toString());
```

Equations

Equations consist of expression like the one above in JModelica. To construct MX objects for these the AST has to be traversed. As shown above, this can be simply done by asking for the children of the expression. Thus, to construct an MX for an expression all that is needed is:

- For a unary expression, e.g. minus, apply the CasADi negation operation to the MX value of the held expression.
- For a binary expression, e.g. addition, apply the CasADi addition operator to the MX value of the held left and right expressions.

A new synthesized attribute toMX is therefore introduced for FExp, from which all other expressions inherit. Note that this attribute is not lazy, it is not necessary that the MX of the expressions are stored since their constituents, the MX for variables, are stored. The other expressions will then override the Java code that computes the value. So, to construct the code needed to be able to call FAddExp.toMX() on the expression in Figure 4.6 the following methods are declared in the aspect:

```
eq FIdUseExp.toMX() = myFV().MX_variable_CasADi();
eq FAddExp.toMX() = getLeft().toMX().__add__(getRight().toMX());
eq FMulExp.toMX() = getLeft().toMX().__mul__(getRight().toMX());
```

Here the methods prefaced with double underscores are calls to CasADi. The FIdUseExp retrieves the MX object that represents the variable it holds.

Functions

Functions pose an additional challenge since MXFunction objects have to be constructed, and they require a single MX expression for every output variable. But since functions may have an arbitrary list of statements, generally there does not exist a single expression for each output variable. For example, in the following function there is no single statement for the output variable out1.

```
function f
    input Real in1;
    input Real in2;
    output Real out1;
    Real internal;
algorithm
    internal := in1;
```

```
    out1 := in2;
    out1 := out1*internal;
end f;
```

Instead the variables are recalculated over and over in the algorithm section.

The transfer of functions problem is twofold, first `MXFunction` must be created for the class that represents functions in the JModelica.org compiler, `FFunctionDecl`. Then these must be called and transformed into function calls for the expressions in which they occur.

Constructing functions

Looking at the function above it is obvious that there exists a single expression for every output variable. All that needs to be done is to make sure that all statements are reflected in the final expression, in this case the final expression for `out1` is obviously `in2*in1`.

The solution was to handle the statements sequentially as they appear in the algorithm section. Each assignment represents an update of a variable, and these updated variables should be reused or reassigned. To handle these updates the idea was to save the updated variables in a list, so that they could be used in new statements. To be able to determine which updated variable that corresponds to the left hand side in the assignments the original variables are saved in another list.

To handle a statement, the variable that corresponds to the left hand side is found in the list with the original variables. The right hand side is then calculated using the updated variables, and the result is saved in the updated variable list. Central to this scheme is the use of the CasADi `substitute` method, as described in Section 2.4.2, which is used to replace the variable occurring in the expressions with their updated values.

As an illustration, to calculate the output expression for the function above we begin by creating the two lists. The current expression list is populated with the variables themselves, since those are the variables current expressions.

Variables	Current expression
in1	in1
in2	in2
out1	out1
internal	internal

The first assignment is `internal:= in1`:

Variables	Current expression
in1	in1
in2	in2
out1	out1
internal	**in1**

The second assignment is `out1 := in2`:

Variables	Current expression
in1	in1
in2	in2
out1	**in2**
internal	in1

The third and final assignment is `out1 := out1*internal`, note how this assignment reuses the values in the current expression list:

Variables	Current expression
in1	in1
in2	in2
out1	**in2*in1**
internal	in1

Thus the function is given by the expression `f(in1,in2) = in2*in1`.

Some additional challenges are posed by the existence of function calls in the statement list. Normal function calls are automatically transferred (as a result of this code), but the function calls with several outputs (see Section 2.2.2, functions) have to be handled separately. The task is to construct the function call for the right hand side, and identify the variables occurring on the left hand side. By choosing the correct output from the function call, these variables may then be updated in the same manner as above.

Transferring function calls

The `MXFunction` constructed above must then be used to construct function calls. In the JModelica.org compiler function calls are a subclass of `FExp`, and for these we implement the `toMX` method. As outlined in Section 2.4.2, the `MX` object representing the call is simply constructed by calling the function with the right arguments.

4.3.3 Filling model

The next step is to use the functionality above to create and fill an instance of the C++ model. This will be achieved by creating a C++ program that creates a JVM, wherein the JModelica.org compiler may reside. The compiler is then asked to flatten a model and produce the final flat class representing it, which is then used to transfer the relevant CasADi objects and other information. As mentioned in Section 2.3, there are actually different compilers for Modelica and Optimica, thus a transfer program for each compiler is created.

The transfer of the different parts of the models is described below. As the Optimica compiler is basically an extended Modelica compiler, no difference between them is made in the presentation of their shared functionality.

Variables

In accordance with the goals presented in chapter 3, we are interested in transferring real variables, i.e. real state, derivative, algebraic, parameter and constant variables. Thus to create variables for the C++ model, the variable's causality and variability has to be determined as well as the attributes that belong to the variable. Another complexity that has to be taken into account is the fact that instances of the class `DerivativeVariable` are required to keep a reference to their state variable.

The flat class conveniently keeps all variables in a single list, and these are looped through. The class that represents variables in JModelica.org provides methods to determine the variability and causality. The class also has methods to access the attributes, and these are of `FExp` type and they implement the `toMX` method, which are simply transferred and added to the attribute map in the `Variable` class.

Derivative and state variables have to be handled specially though. When looping through the list of variables, derivative variables are ignored. When a state variable is encountered, it is asked for its derivative variable, and both variables are constructed at the same time.

Equations

The flat class keeps the initial and model equations in separate lists. These lists contain instances of the `FAbstractEquation` class. Two subclasses have to be considered, `FEquation` and `FFunctionCallEquation`.

`FEquation` basically holds expressions for its left and right hand side, and `MX` expressions for these may be automatically transferred (even those containing function calls).

`FFunctionCallEquation`, as described in Section 4.3.2 under functions, keeps a list with objects that contain `FExp` for its left hand side, and a function call in its right hand side. The function call is automatically transferred using `toMX`, and an `MX` object is created for the list of left hand side objects.

Functions

In Section 4.2.4 the function representation, `ModelFunction`, for the C++ model was described. It keeps the `MXFunction`, and a list of statements and variables.

The flat class saves all of the function declarations in a list, and these are looped through. Every function declaration keeps a list of statements, and these can be transferred in the same way as equations.

`ModelFunction` keeps the variables sorted into the categories ins, internal and outs. These may be retrieved from the function declarations.

Optimica

To transfer Optimica classes, some extra steps beyond the transfer of Modelica models are required. The Lagrange and Mayer terms are kept as expressions in

the flat class, and are thus easily transferred and stored in `OptimizationProblem`. Similarly for the start and final time attributes. To transfer the new variable attributes, the list of attributes to transfer is extended.

Next path constraints are transferred. The flat class keeps lists with the different path constraints, i.e. greater than, less than and equal to constraints. These constraints keep, similar to equation, `FExp` for the left and right hand side, and `MX` can be transferred for them. Instances of the `Constraint` class are created for each constraint, and stored in a list in `OptimizationProblem`

4.4 Testing

A collocation method for solving OCPs in Python using CasADi, using implicit Euler integration, was provided by my advisor Fredrik. It was translated to C++ and extended to solve OCPs represented by the `OptimizationProblem` class. The method uses the `IPOPT` solver. To visualize the solutions, a method that creates Python scripts for plotting the output from the solver was created. A method that saves the solution trajectories to a text file was also implemented.

A test that compares solution trajectories from the optimization method above to trajectories obtained from a similar optimization method built into JModelica.org, using the root mean square norm, was implemented.

Chapter 5

Benchmarks

Below some benchmarks are described. First a Van der Pol based optimization problem is solved using JModelica.org and the implementation, and the results are compared graphically and numerically. Then the different ways to solve OCPs in JModelica.org are timed, i.e. using JMUs or XML export/import with `SymbolicOCP`, and compared to the implementation. Provided is also an example of a printout of a model, and a result showing that the implemented function handling can be more stable than code-unit compilation.

5.1 Solution comparison

A Van der Pol based optimization problem is solved. This problem is described by:

Minimize:

$$\int_0^1 \frac{(x_1(t)^2 + x_1(t)^2 + u(t)^2)}{10} \mathrm{d}t$$

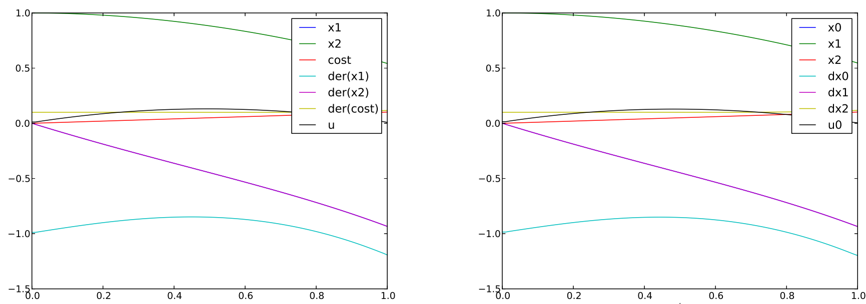
Subject to:

$$\begin{aligned} &\dot{x}_1(t) = (1 - x_2(t)^2) \cdot x_1(t) - x_2(t) + u(t) \\ &\dot{x}_2(t) = x_1(t) \\ &u(t) \leq 0.75, \quad \forall t \in [0, 1] \end{aligned}$$

The corresponding Optimica class can be written as:

```
optimization vdp (objectiveIntegrand = cost,
                      startTime = 0,
                      finalTime = 1)

  // The states
  Real x1(start=0,fixed=true);
  Real x2(start=1,fixed=true);

  // The control signal
  input Real u;

  function cost_function
    input Real in1;
    input Real in2;
    input Real in3;
    output Real out;
  algorithm
    out := in1^2  + in2^2 + in3^2;
    out := out/10.0;
  end cost_function;

  Real cost(start=0,fixed=true);

equation
  der(x1) = (1 - x2^2) * x1 - x2 + u;
  der(x2) = x1;
  der(cost) = cost_function(x1,x2,u);
constraint
   u<=0.75;
end vdp;
```

This problem was solved using the Optimica compiler, which creates a JMU code-unit. Options were given to the solver so that the algorithm used was similar to the one implemented in this thesis. The problem was also solved using the implemented scheme. Both optimization methods use the solver IPOPT. The resulting trajectories are shown in Figure 5.1.



(a) JModelica.org solution to Van der Pol optimization problem, using JMU.

(b) Solution to transferred and then solved Van der Pol optimization problem.

Figure 5.1: Comparison of solutions to a Van der Pol optimization problem using JModelica.org and the implementation.

The root-mean-square norm of the difference between the trajectories was computed, and it had a magnitude of 10^{-5}.

5.2 Timing

The time it took to use the two different ways to solve an Optimica class in JModelica.org were compared to the implemented scheme. The following Optimica class was solved:

```
optimization SimpleDAEOpt (objectiveIntegrand=a^2 + abs(b) + u, finalTime=10)
    Real a(start=5.0);
    Real b(start=1.0,fixed=true);
    input Real u;
equation
    der(a) =  -u*b^4;
    b = 1.0+u^2;
constraint
    a <=  0.00;
end SimpleDAEOpt;
```

The results tabulated:

Solution scheme	time [s]
JMU code-units	20.8
XML export / import	7.2
Implemented scheme	9.3

For JMUs the measured time starts when JModelica.org is ordered to compile a JMU for the Optimica class, and ends when it has used solvers to solve the resulting JMU. For XML export/import the measured time starts when JModelica.org is ordered to construct an XML-file for the Optimica class, and ends when it has been imported and solved with the CasADi import module. For the implemented scheme the measured time starts when the transfer program is ordered to transfer the Optimica class, and ends when it has been transferred and solved.

5.3 Printing

The result of printing the Van der Pol Optimica class listed in Section 5.1 is shown below. Here line breaks were manually added in the statements list for the function, since the lines would be too long to view in this document otherwise.

```
Model contained in OptimizationProblem:

------------------------------- Variables -------------------------------

MX(startTime), attributes:
    bindingExpression = MX(Const<0>(scalar))

MX(finalTime), attributes:
    bindingExpression = MX(Const<1>(scalar))

MX(der_x1)
MX(der_x2)
MX(der_cost)
MX(x1), attributes:
    fixed = MX(Const<1>(scalar))
    start = MX(Const<0>(scalar))

MX(x2), attributes:
    fixed = MX(Const<1>(scalar))
    start = MX(Const<1>(scalar))

MX(cost), attributes:
    fixed = MX(Const<1>(scalar))
    start = MX(Const<0>(scalar))

MX(u)
------------------------------- Functions -------------------------------

ModelFunction : function("vdp.cost_function")
    Ins:
```

```
        MX(funcVar_in1)
        MX(funcVar_in2)
        MX(funcVar_in3)
    Internal:
    Outs:
        MX(funcVar_out)
    Statements:
        MX(funcVar_out) = MX(((pow(funcVar_in1,Const<2>(scalar))+
                               pow(funcVar_in2,Const<2>(scalar)))+
                               pow(funcVar_in3,Const<2>(scalar))))
        MX(funcVar_out) = MX((funcVar_out/Const<10>(scalar)))
------------------------------- Equations -------------------------------

 -- Initial equations --
MX(x1) = MX(Const<0>(scalar))
MX(x2) = MX(Const<1>(scalar))
MX(cost) = MX(Const<0>(scalar))
 -- DAE equations --
MX(der_x1) = MX(((((Const<1>(scalar)-pow(x2,Const<2>(scalar)))*x1)-x2)+u))
MX(der_x2) = MX(x1)
MX(der_cost) = MX(function("vdp.cost_function").call([vertcat(x1,x2,u)]){0})

-- Optimization information  --

Start time = MX(Const<0>(scalar))
End time = MX(Const<1>(scalar))
-- Constraints --
MX(u) <= MX(Const<0.75>(scalar))
-- Lagrange term --
MX(cost)
-- Mayer term --
MX(Const<0>(scalar))
```

5.4 Stability

The following Optimica class, which is mathematically equivalent to the class `SimpleDAEOpt` listed in Section 5.2, caused the solver to crash when JMU was used to solve it. It does not cause the implemented scheme to crash however.

```
optimization Rewrites (objectiveIntegrand=a^2 + abs(b) + u, finalTime=10)
    Real a(start=5.0);
    Real b(start=1.0,fixed=true);
    input Real u;
    function f
      input Real a;
```

```
      output Real b;
    algorithm
      b := a^2;
    end f;
    function f2
      input Real a1;
      input Real a2;
      output Real b;
      Real c;
    algorithm
      c := a1;
      c := c*a1;
      b := c+a2*f(1.0);
      c := c*2.0;
      c := c /2.0;
      b := b - c;
      b := b + c;
      b := b+2;
    end f2;
equation
    der(a) =  -u*b^4;
    b*f(1.0) = f(1.0)+f2(u,a)-a-2;
constraint
    a <=  0.00;
end Rewrites;
```

Chapter 6

Discussion

The goal for this thesis was to a lay a foundation for a new way for JModelica.org to handle models, using CasADi. There were some specific goals, to transfer and represent translated Modelica and Optimica code that may contain functions, and there were some general goals, that the resulting software is efficient, easy to understand and extensible. How well these goals have been achieved is discussed below, and also future work and other considerations.

6.1 Goal evaluation

6.1.1 Specific goals

The developed model is able to represent the implicit DAE system from Modelica models by using CasADi. To do this, it holds variables and equations. Functions may be part of these equations. We note that the function representation, using CasADi's built in function class, is more stable than the code-based one found in JMUs in at least a few cases. The model also allows for the formulation of optimization problems. This is done in a separate class that holds an instance of the DAE system, as well as other information such as cost functions and constraints that are associated with Optimica classes.

Functionality has been implemented to use the JModelica.org compiler to translate Modelica and Optimica code and transfer this information to the model representation. This was achieved by extending the Java based JModelica.org compiler so that it is able to create the required CasADi objects, and by connecting the JModelica.org compiler with C++ so that this information could be transferred and used to create an instance of the C++ based model representation.

All developed classes support printing functionality. To make the printing of functions as understandable as possible the decision was made to transfer a more complete representation of the function, its internal variables and statements, than was needed to obtain a correct mathematical representation.

That the model representation and transfer were constructed in a correct manner was verified by implementing test methods that solved the transferred optimization problems. The output from these solvers showed results that were very similar to those obtained using the corresponding optimization algorithm found in JModelica.org.

The time it takes to solve an optimization problem compares well to the existing ways, using JMUs and XML export/import. The implemented scheme was about twice as fast as using JMU code-units, for simple classes at least. A large part of the difference was observed to be due to the creation of the code units themselves. Compared to using XML export/import the scheme was a bit slower, and here the difference was observed to be mostly due faster solver performance, which is probably because the XML export/import (with the CasADi based `SymbolicOCP`) uses `SX` instead of `MX`.

6.1.2 General goals

Great care has been taken to make the model representation as efficient as possible. For example, to access variables the BOOST library has been used to utilize methods with lower time complexity than traditional ones. Other examples are the use of lazy classification, which allows the variables to be sorted a single time after the model has been filled, and not moving objects by value, instead opting to use pointers. Lessons from what works well in JModelica.org have also been utilized; an example here is the use of a single variable vector. The model is simple and has a representation that is close to the mathematical description, which makes it understandable and extendable.

Before building the transfer mechanics tests were performed to determine the fastest way to populate the model with the right CasADi objects, which was to let JModelica.org create them. This approach has been used to transfer all CasADi objects. For efficiency, the use of lazy attributes has been avoided where possible. We note that many parts of the transfer were easy to implement since the JModelica.org AST has a structure with many similarities to the implemented model.

As for the integration with CasADi the use of `MX` was chosen over `SX`. The reason is that this is more general and allows for a convenient handling of functions, on the other hand `SX` is faster and more intuitive. Before we choose this approach it was investigated whether this approach could work with the extended hybrid scheme that we aim to support in the future. A scheme was developed by my advisor Toivo and proposed to the CasADi developers who deemed it realistic.

Thus we argue that the general goals are fulfilled. It would have been good if the efficiency goals were tested more though, using e.g. some resource consumption test, but there was not enough time to implement any.

6.2 Future work

It should be easy to add support for many parts of Modelica and Optimica, e.g. point constraints and arrays, but other additions will probably prove more difficult. To provide support for hybrid DAEs it will probably be necessary to extend CasADi.

To make use of the interactivity provided by CasADi, interfaces to Python will need to be implemented.

An attractive possibility is integration with XML-based transfer. By implementing XML import JModelica.org will not need to be activated at the same time as model, reducing the burden on working memory.

6.3 Other considerations

CasADi is a tool under active development, and its inner workings, interfaces and supported functionality will probably change over time. As such this implementation might need to be updated to comply with new versions of CasADi.

Bibliography

[1] "Modelica home page." `https://www.modelica.org/`. Accessed: 2013-04-29.

[2] P. Fritzson, *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1.* Wiley, 2004.

[3] "Jmodelica.org homepage." `http://www.jmodelica.org/`. Accessed: 2013-06-12.

[4] J. Andersson, J. Åkesson, and M. Diehl, "Dynamic optimization with CasADi," in *51st IEEE Conference on Decision and Control*, (Maui, Hawaii, USA), Dec. 2012. Accepted for publication.

[5] J. Åkesson, *Languages and Tools for Optimization of Large-Scale Systems.* PhD thesis, Regler, Nov. 2007.

[6] M. M. Tiller, *Introduction to Physical Modeling with Modelica.* Dordrecht, The Netherlands: Kluwer Academic Publishers Group, 2001.

[7] P. Fritzson and P. Bunus, "Modelica-a general object-oriented language for continuous and discrete-event system modeling and simulation," in *Simulation Symposium, 2002. Proceedings. 35th Annual*, pp. 365–380, IEEE, 2002.

[8] "Modelica specification 3.2." `https://www.modelica.org/documents`. Accessed: 2013-06-14.

[9] "Jastadd homepage." `http://jastadd.org/web/`. Accessed: 2013-06-12.

[10] "Abstract syntax tree, wikipedia." `http://en.wikipedia.org/wiki/Abstract_syntax_tree`. Accessed: 2013-06-12.

[11] J. Andersson, J. Åkesson, and M. Diehl, "CasADi – A symbolic package for automatic differentiation and optimal control," in *Recent Advances in Algorithmic Differentiation*, vol. 87 of *Lecture Notes in Computational Science and Engineering*, pp. 297–307, Springer Berlin Heidelberg, 2012.

[12] "Casadi home on github." `https://github.com/casadi/casadi/wiki`. Accessed: 2013-04-23.

[13] "Automatic differentiation, wikipedia." http://en.wikipedia.org/wiki/Automatic_differentiation. Accessed: 2013-04-23.

[14] "Casadi user guide." https://github.com/casadi/casadi/wiki/users_guide. Accessed: 2013-06-12.

[15] T. Blochwitz, M. Otter, M. Arnold, C. Bausch, C. Clauß, H. Elmqvist, A. Junghanns, J. Mauss, M. Monteiro, T. Neidhold, *et al.*, "The functional mockup interface for tool independent exchange of simulation models," in *Modelica'2011 Conference, March*, pp. 20–22, 2011.

[16] "Jmi model interface." http://www.jmodelica.org/api-docs/jmi/group__Jmi.html. Accessed: 2013-06-13.

[17] R. Parrotto, J. Åkesson, and F. Casella, "An xml representation of dae systems obtained from continuous-time modelica models," in *3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools,(Oslo, Norway)*, pp. 91–98, Citeseer, 2010.

[18] F. Magnusson, "Collocation methods in JModelica.org," Master's Thesis ISRN LUTFD2/TFRT--5892--SE, Department of Automatic Control, Lund University, Sweden, Feb. 2012.

[19] "Boost homepage." http://www.boost.org/. Accessed: 2013-06-08.

[20] "Oracle jni documentation." http://docs.oracle.com/javase/6/docs/technotes/guides/. Accessed: 2013-06-06.

[21] "Jcc homepage." http://lucene.apache.org/pylucene/jcc/features.html. Accessed: 2013-06-06.

[22] "Swig homepage." http://www.swig.org/. Accessed: 2013-06-06.