JMODELICA—AN OPEN SOURCE PLATFORM FOR OPTIMIZATION OF MODELICA MODELS

J. Åkesson[1,2], M. Gäfvert[2], H. Tummescheit[2]
[1]Lund University, Sweden, [2]Modelon AB, Sweden

Corresponding author: J. Åkesson, Modelon AB, Beta-building Scheelevägen 17,
SE-223 70 Lund, Sweden, johan.akesson@modelon.se

Abstract. Optimization is becoming a standard methodology in many engineering disciplines to improve products and processes. The need for optimization is driven by factors such as increased costs for raw materials and stricter environmental regulations as well as a general need to meet increased competition. As model-based design processes are being used increasingly in industry, the prerequisites for optimization are often fulfilled. However, current tools and languages used to model dynamic systems are not always well suited for integration with state of the art numerical optimization algorithms. As a result, optimization is not used as frequently as it could, or less efficient, but easier to use, algorithms are employed.

This paper reports a new Modelica-based open source project entitled JModelica, targeted towards dynamic optimization. The objective of the project is to bridge the gap between the need for high-level description languages and the details of numerical optimization algorithms. JModelica is also intended as an extensible platform where algorithm developers, particularly in the academic community, may integrate new and innovative methods. In doing so, researchers gain access to a wealth of industrially relevant optimization problems based on existing Modelica models, while at the same time facilitating industrial use of state of the art algorithms. The JModelica project rests upon three pillars, namely a language extension of Modelica for optimization entitled Optimica, software tools, and applications. In this paper, these three topics will be highlighted.

1 Introduction

Optimization is becoming a standard methodology in a wide range of engineering domains, ranging from thermo-fluid applications to vehicle systems. Typical targets for optimization are energy consumption, raw material utilization, and production yield. This trend is largely driven by increasingly competitive markets, the need for product diversification to meet customer demands, and environmental regulations.

More often than not, systems to be optimized are complex and dynamic. Such problems offer several challenges at different levels. Much effort has been devoted to encapsulating expert knowledge in model libraries encoded in domain specific languages such as VHDL-AMS [21] and Modelica [28]. While such model libraries have been primarily intended for simulation, it is desirable to enable also other usages, including optimization. From a user's perspective, it is desirable that the optimization specification is expressed in a high-level language in order to provide a comprehensive description both of the dynamic model to be optimized and of the optimization problem. Another aspect that requires attention is that of enabling flexible use of the wealth of numerical algorithms for dynamic optimization, based on the high-level descriptions specified by the user.

Several common engineering tasks are conveniently cast as optimization problems. This includes parameter estimation problems to obtain models that match plant data, design optimization for improving product performance, and controller parameter tuning. In addition, dynamic optimization is a key to implementing for example model predictive controllers and receding horizon state estimators.

It is typical that numerical algorithms for dynamic optimization is written in C or Fortran. Often, the user is required to encode the dynamic model and the optimization specification in the same languages. While C and Fortran enables efficient compilation to executable code, such languages are not well suited for encoding of dynamic models and optimization problems. In particular, it is difficult to write the code in a modular way that enables reuse. This observation was made several decades ago in the context of modeling and simulation and resulted in high-level modeling languages, including ACSL and later Omola, [3], VHDL-AMS [21], and Modelica [28]. See [5] for a comprehensive overview. Currently, there are several commercial simulation tools available on the market, e.g., gPROMS [27] and Dymola [10]. For dynamic optimization, the situation is somewhat different. To some extent the situation is similar to that of simulation. It is well known that some systems require stiff implicit solvers in order to be accurately integrated while other systems may well be integrated using less sophisticated and explicit solvers. For dynamic optimization, however, the situation is more complicated, and in particular different algorithms typically target different classes of systems, such as linear systems, non-linear ordinary differential equations (ODE), differential algebraic equations (DAE) and hybrid systems. It is also typical that tools supporting dynamic optimization are limited to a particular application domain and/or a particular numerical algorithm.

This contribution reports a new Modelica-based open source initiative targeted at dynamic optimization entitled

JModelica. The JModelica platform builds upon the well established modeling language Modelica, and provides means to formulate, using the language extension Optimica, optimization problems. The use of an established modeling language is natural for two reasons. Firstly, it eliminates the need to reinvent existing technology, and secondly it enables use of a wealth of existing model libraries for optimization purposes.

The JModelica platform is intended to be released as open source software during 2009. As such, JModelica holds the potential of serving as a vehicle for propagating methods and algorithms developed in the academic community into industrial use. In particular, JModelica provides interfaces particularly well suited for integration with various optimization algorithms. In fact, one of the main motivations for the JModelica project is to bridge the gap between the engineering need for high-level description frameworks and the often cumbersome interface of state of the art numerical algorithms for dynamic optimization. This strategy increases the efficiency of the engineering design process for two reasons. Firstly, the user is enabled to use dedicated high-level languages for specifying the optimization problem which in turn promotes understanding and reusability. Secondly, the user may easily experiment with different optimization algorithms and then select the algorithm that is the most appropriate for the particular problem at hand.

The JModelica platform and Optimica has been used in projects both in industry and in academia. In [17], the start-up problem for a plate reactor is considered. The problem is challenging both due to the size of the plant model (approx. 130 variables and states) and due to non-linear and unstable dynamics. In this project, JModelica and Optimica facilitated the highly iterative procedure of obtaining satisfactory solution profiles. In a recent project reported in [4], optimal control of a car is considered. This project is also highlighted in Section 5.

The paper is outlined as follows. In Section 2, overviews of the Modelica language and of the field of dynamic optimization are given. The Optimica extension is described in Section 3 and in Section 4 an overview of the JModelica open source platform is given. An application example is given in Section 5 and the paper ends with a summary in Section 6.

2 Background

2.1 Modelica

The JModelica platform is based on the modeling language Modelica, [28, 16]. Modelica is designed for modeling of complex heterogeneous physical systems, including mechanical, electrical, thermal, and chemical systems. The Modelica language is object-oriented, and supports classes and inheritance as well as component-based modeling. Components, in turn, may be connected through acausal physical interfaces that are modeled explicitly. This particular feature of Modelica is distinguishing in comparison to the block-based modeling formalism where the signal directions need to be determined by the modeler. Behavior in Modelica is specified by means of declarative algebraic and differential equations. Also in this respect Modelica offers a user friendly input format, since equations can be entered on their natural form without the need to solve for the derivatives, which is otherwise common. Modelica also supports modeling of hybrid phenomena such as friction and backlash as well as state machines useful for control system modeling.

The Modelica language is open and is developed by the non-profit organization Modelica Association, [28]. In addition, a freely available standard library is available which includes models within the domains multi-body dynamics, electronics and electrical machines, heat transfer, fluid media, and control systems. There are several Modelica tools and model libraries on the market, both commercial and free.

2.2 Dynamic optimization

The field of dynamic optimization is represented by a large body of research and numerous applications. The theory originates from calculus of variations, which is a branch of mathematics. The theoretical development was driven in the 50s and 60s by the space race, and two notable contributions were made by Pontryagin, [26], and Bellman, [7]. Pontryagin's maximum principle states the optimality conditions for an open loop optimal control problem, whereas Bellman's contribution, dynamic programming, is concerned with finding the closed loop solution of a similar problem. While both the maximum principle and dynamic programming has been used successfully in many applications, they are notoriously hard to apply important classes of problems. These include large-scale non-linear problems. In addition, inequality constraints are difficult to manage with these methods.

In response to the mentioned difficulties, a new family of methods has emerged, referred to as direct methods. In the direct methods, the original continuous time optimal control problem is translated (transcribed) into a discrete time approximation. The resulting finite dimensional problem is then addressed by means of non-linear programming (NLP) algorithms. There are two main directions within the class of direct methods; sequential methods and simultaneous methods. A sequential method is based on the shooting principle. In a first step, the system dynamics is integrated, possibly along with the sensitivity equations in order to obtain gradients, to evaluate the cost function and the constraints. In a second step, an NLP algorithm computes new values for the tuning parameters and the procedure is repeated. See [30] for an overview. The simultaneous methods, in contrast, are based on orthogonal

```
model VDP
  Real x1(start=0);
  Real x2(start=1);
  input Real u;
equation
  der(x1) = (1-x2^2)*x1 - x2 + u;
  der(x2) = x1;
end VDP;
```

Listing 1: A Modelica model of a van Der Pol oscillator.

```
optimization VDP_Opt (objective=cost(finalTime),
            startTime=0,
            finalTime(free=true, initialGuess=1))
① VDP vdp(u(free=true,initialGuess=0.0));
② Real cost (start=0);
equation
③ der(cost) = 1;
constraint
④ vdp.x1(finalTime) = 0;
⑤ vdp.x2(finalTime) = 0;
⑥ vdp.u >= -1;
⑦ vdp.u <= 1;
end VDP_Opt;
```

Listing 2: An Optimica optimization specification based on the van Der Pol Oscillator.

collocation, where both the state and control variable spaces are discretized, typically using polynomials. This strategy often renders very large NLP problems, which require efficient solvers capable of exploring the inherent sparsity in such problems. See [9, 8] for an overview of simultaneous methods. In addition, a class of methods referred to as multiple shooting methods, [23, 24], has won widespread use in many applications. A multiple shooting strategy can be viewed as a combination of a sequential and a simultaneous method in that the optimization interval is divided into distinct elements that are integrated independently.

In the context of the JModelica project, attention has been given primarily to direct methods. This choice follows from the observation that most Modelica models are of large scale and that inequality constraints often play an important role in realistic applications. In particular, a simultaneous method is currently supported by the JModelica platform. However, other methods may be supported in the future, in line with the objectives of the JModelica project.

3 Optimica

Modelica does not support convenient formulation of dynamic optimization problems. This is quite natural, since Modelica is primarily designed for description of simulation models. Never the less, constructs essential for optimization are lacking, e.g., the notion of a cost function, constraints and specification of the optimization interval. In [1], a language extension of Modelica, entitled Optimica, was presented. The objective of Optimica is to provide means for the user to express optimization problems based on Modelica models. This involves specification of information at three levels. Firstly, the canonical description of the optimization problem, including the cost function, what parameters and variables to tune, the optimization interval, and the constraints. Secondly, the method of transcription needs to be specified. Typically, it is necessary to tailor the method of solution to a particular problem in order to achieve desirable solution properties and convergence speed. Thirdly, control parameters to solvers, such as for examples tolerances usually needs to be specified. Following the design principles of the Modelica language, information related to the first category is specified using dedicated language elements, whereas the transcription method and algorithm parameters are specified using annotations. This strategy has the benefit of making a clear distinction between the encoding of the actual problem formulation, and the method used to solve the problem.

3.1 The anatomy of an Optimica specification

The Optimica extension is discussed in detail [1]. In this paper, a brief overview of Optimica is given and the extension is illustrated by means of an example. We consider the following dynamic optimization problem:

$$\min_{u(t)} \int_0^{t_f} 1\, dt \tag{1}$$

subject to the dynamic constraint

$$\begin{aligned} \dot{x}_1(t) &= (1-x_2(t)^2)x_1(t) - x_2(t) + u(t), \quad & x_1(0) &= 0 \\ \dot{x}_2(t) &= x_1(t), & x_2(0) &= 1 \end{aligned} \tag{2}$$

and

$$\begin{aligned} x_1(t_f) &= 0 \\ x_2(t_f) &= 0 \\ -1 \leq & u(t) \leq 1 \end{aligned} \tag{3}$$

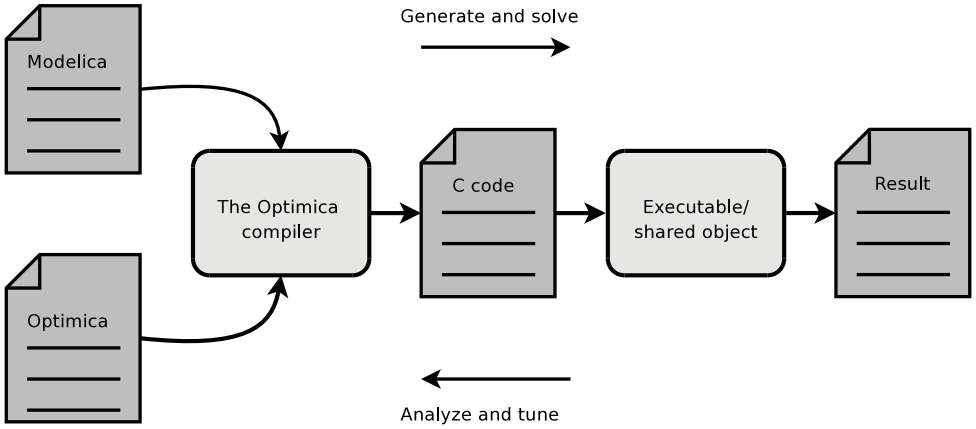

Figure 1: A typical working cycle for dynamic optimization.

The dynamic model (2) of the problem is a van Der Pol oscillator, and the optimization problem corresponds to bringing the system from initial conditions $x_1(0) = 0$, $x_2(0) = 1$ to the origin in minimum time. In addition, the transition is to be performed with limited control authority.

A Modelica model corresponding to the dynamic system (2) is given in Listing 1. Based on this model, an Optimica specification can be formulated, see Listing 2. Since Optimica is an extension of Modelica, language elements valid in Modelica are also valid in Optimica. In addition Optimica also contains new constructs not valid in Modelica.

The Optimica program corresponding to the van Der Pol example can be seen in Listing 2. In order to specify an optimization problem in Optimica, the new specialized class `optimization` is used. Inside such a class, Optimica constructs, as well as Modelica constructs may be used. An instance of the `VDP` model is created by declaring a corresponding component, ①. In order to express that the input `u` is to be tuned in the optimization, the Optimica-specific variable attribute `free` is set to `true`, and in addition, an initial guess for `u` is provided. In order to define the cost function, a variable, `cost` ②, is introduced along with a defining equation, ③. Further, the constraints are given in the `constraint` section, which is a new Optimica construct. In this section, ④–⑤ correspond to the terminal constraints, whereas ⑥–⑦ correspond to the control variable bounds. Notice how the value of a variable at a particular time instant is accessed using an Optimica-specific function call-like syntax. `finalTime` is a built-in variable of the specialized class `optimization` and is used to refer to the time at the end of the optimization interval. Finally, the objective and the optimization interval is specified, ⑧. The construct introduced in Optimica to meet this end can be viewed as built-in class attributes which are given values through class arguments. Here the variable representing the cost function is bound to the built-in class attribute `objective` and it is specified that `finalTime` is to be free in the optimization.

The Modelica and Optimica specifications are then typically translated by a compiler into a format suitable for compilation with a numerical solver in order to obtain the solution.

4 The JModelica open source platform

The JModelica platform consists of a collection of software modules, including two compiler front-ends for Modelica and Optimica respectively, a code generation back-end for C, a run-time library in C, a simultaneous optimization algorithm, and a library for integration with Python. Together, these pieces of software form a complete tool chain for formulating and solving dynamic optimization problems based on Modelica models. The foundation of the JModelica platform in its current form was presented in [1]. As part of this PhD project, a prototype compiler was developed, which is currently being further developed into the JModelica open source platform. The project is scheduled for public release during 2009.

The data flow of the JModelica tool chain is illustrated in Figure 1. It is typical that the solution of dynamic optimization problems require multiple iterations, where the cost function, the constraints, the transcription method, and even the model are refined in order to obtain satisfactory results. Each cycle contains the steps *i)* modifications to the source codes, *ii)* generation of C code and compilation to obtain an executable, and *iii)* solution of the optimization problem in order to obtain the result. The result then typically needs to be analyzed and the optimization formulation adjusted. The use of high-level description languages then relieves the user of the cumbersome and error-prone task of encoding the model and optimization formulations in less suitable languages. In effect, the focus of the design process is shifted from *encoding* of the problem into *formulation* of the problem, which translates into more efficient design processes.

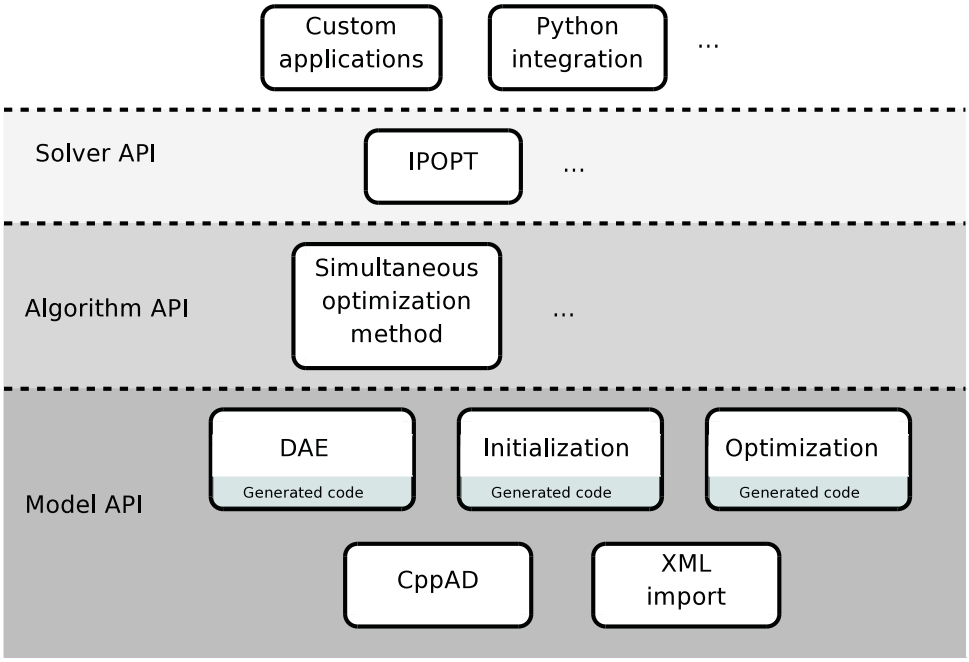

Figure 2: Architecture of the model and optimization interfaces.

4.1 Development environment and compiler design

The compiler front-ends and the code generation module are developed in the compiler construction framework JastAdd [19]. The name JastAdd is used to refer both to the JastAdd language, which is essentially an extension of Java and to the complementing JastAdd tool. In JastAdd, several concepts including object orientation, aspect orientation, and reference attributed grammars are combined into a framework that is suitable for development of compilers. A particular feature of JastAdd that makes it well suited for the JModelica project is that it supports construction of modularly extensible compilers, both at the language level and at the implementation level. As a consequence, it is straight forward to develop and maintain a core compiler, in this case a pure Modelica compiler, while one or several extensions are developed independently. Accordingly, the Optimica compiler is developed as a fully modular compiler that builds on the core Modelica compiler. JastAdd is released under an open source license and is available at [11]. The design of the core compiler front-end is described in [2]. In this paper, details of the implementation is discussed, including name and type analysis as well as a Modelica flattening algorithm. The Optimica extension is discussed in [18], where the extensibility capabilities of the core compiler is described and analyzed.

The Optimica compiler consists, conceptually and practically of a front-end and a code generation back-end. The task of the compiler front-end is to read Modelica and Optimica sources, perform code analysis, error checking and finally produce a flat representation of the source code. The analysis step includes classical compiler tasks such as name analysis (binding identifiers to declarations), type analysis (verify type correctness of a program). In the flattening step, component and inheritance structures are resolved and a flat representation consisting essentially of a set of variables and a set of equations is derived. The flat representation can be viewed as an intermediate format that can then be further analyzed and transformed.

After flattening of the Modelica and Optimica specifications, the code generation back-end is invoked in order to produce C code. The generated C code provides a low level interface to the model and the optimization formulations, and is suitable for integration with numerical algorithms. In addition, model meta data files in XML format are generated.

4.2 Model and optimization interface

The C code generated by the Optimica compiler is intended to be integrated with numerical algorithms. To meet this end, an Application Program Interface (API) consisting of a set of C functions is provided. The architecture of the JModelica API is depicted in Figure 2. As can be seen, the software is structured into three main parts, namely the Model API, the Algorithm API, and the Solver API.

The Model API has three parts, namely the DAE interface which gives access to the DAE system, the initialization interface which represents the equations that must be solved in order to consistently initialize the DAE, and finally the optimization interface which provides access to e.g., the cost function and the constraints. The division of the Model API is motivated by the fact that not all parts of the interface may be provided in all cases. For example, it may be that only the DAE initialization part is provided if a static problem is to be solved. The Model API

functions encapsulate the generated code, and also provide means to access the XML meta data files that are generated along with the C code. In order to provide derivatives of high accuracy, which is critical for many optimization algorithms, an open source package providing automatic differentiation of functions, CppAD [6], has been integrated in the Model API. Based on the CppAD functionality, the Model API provides functions for evaluation of Jacobians of the functions included in the DAE, initialization, and optimization interfaces. In addition, sparsity information for the Jacobians may be computed.

The Model API is designed to be generic and may be used as a basis for various purposes, not only optimization. Rather, the Model API is a suitable access point for developers interested in integrating their own algorithms into the JModelica platform. Algorithms are implemented in the Algorithm part of the JModelica API. In the current version of JModelica, a simultaneous optimization algorithm is implemented. The method is based orthogonal collocation using Lagrange polynomials on Radau points, see e.g., [9]. The algorithm is implemented in C and utilizes the functions provided by the Model API. In essence, the simultaneous optimization algorithm transcribes the continuous time optimization problem into a large algebraic NLP problem. Accordingly, the simultaneous optimization module offers an interface to the resulting NLP, which in turn may be solved by an NLP algorithm. In JModelica, the open source solver IPOPT, [31], is used. This solver is interfaced to the simultaneous optimization interface in a separate module in the Solver API. This design facilitates integration of alternative solvers as well as algorithms.

It is worth mentioning that here is a parallel effort within the European ITEA2 MODELISAR project to define the Functional Model Interface (FMI), which is a new open interface standard for run time co-simulation of Modelica models. FMI has similar objectives as the JModelica Model API, but is intended for simulation only and does not address the specific needs for optimization applications.

4.3 Model metadata

Model metadata is collected in XML files that follows JModelica W3C XML Schema definitions. These XML files contain information on model versions, model structure, variable names and types, etc. These metadata files are generated by the compiler in parallel to the model C-code. Parameter sets are also maintained and stored as XML files. The XML format makes it easy to maintain and process model information with standard tools, and makes it convenient to generate model documentation with XSLT transformations on the XML files.

There are several ongoing efforts to standardize Modelica model metadata with XML Schema definitions. It is likely that these will merge into one single standard in the future.

4.4 Python integration

Python, see [15], is chosen as scripting language for JModelica applications. Python is a natural choice since it is a free open-source highly efficient and mature scripting language with strong support in the scientific community. Packages such as NumPy [25] and SciPy [13], and bindings to state-of-the art numerical codes implemented in C and Fortran make Python a convenient glue between the components in the JModelica toolchain. IPython [12] with the visualization package matplotlib [20] and the PyLab mode offer an excellent interactive numerical environment.

The JModelica Python integration is done via the Model API and dynamically linked (shared) libraries. The model C-code is compiled into shared libraries with the signature of the Model API, and then loaded into Python using the ctypes package [14]. This approach gives a convenient and direct integration of the compiled model C-code into Python, with less programmatic overhead compared to integration based on extensions using the Python C/C++ API. The Model API shared libraries can also be wrapped into generic user-friendly Python classes with methods and properties to access the model equations and properties. In combination with, e.g., SUNDIALS [22] with Python bindings [29] this forms a powerful toolset that can be used for shooting-based optimization methods.

5 Application example

This section briefly summarizes a recent application of JModelica and Optimica to evaluate the maneuverability limits of road vehicles with different steering and propulsion configurations. The full study is published in [4]. The study considers a concept car with individual steering, braking, driving, and normal load distribution on each of the four wheels. The car is encoded in Modelica by a 6-degrees of freedom vehicle model with longitudinal, lateral, and yaw in-plane motion, and vertical, roll, and pitch out-of-plane motion, see Figure 3 (left). The tires are described by non-linear mappings from normal load and applied longitudinal force to lateral force, that represent the force production characteristics and friction limitations. With additional constraints on actuator magnitude and rate limitations this vehicle can represent standard front wheel steer (FWS), an all wheel steer (AWS), all wheel drive (AWD), rear wheel drive (RWD), and front wheel drive (FWD) configurations. A maximum displacement maneuver is now defined, with the objective to move the car sideways as much as possible, subject to maintained directional stability and entry and exit conditions on position and velocity. The maneuver relates to the ability of the car to safely respond to obstacle avoidance maneuvers by the driver or an active safety system. The maximum displacement maneuver is defined in Optimica as a cost function with a set of equality and inequality constraints. The

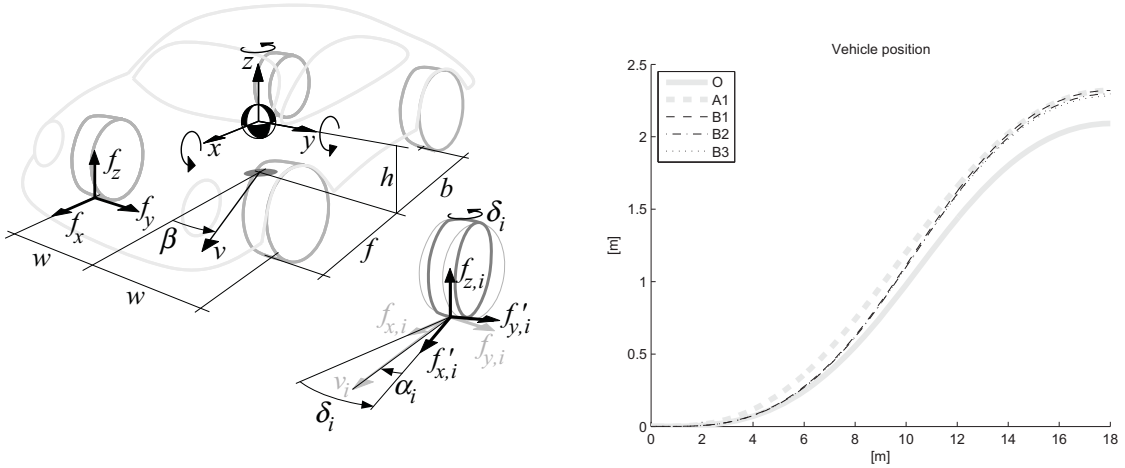

Figure 3: Left: Vehicle model with notation; Right: Resulting optimal vehicle path for maximum displacement maneuver (O - FWS, A1 - AWS, B1 - AWD, B2 - FWD, B3 - RWD).

performance difference between different vehicle configurations can now be assessed by solving the corresponding optimization problems. A snapshot of the result is shown in Figure 3 (right), where the displacement trajectories are visualized for some different configurations. The results illustrate fundamental performance limitations in the vehicle configurations and give guidance in development decisions on future vehicle concepts.

6 Summary

This paper reports a novel Modelica-based open source initiative targeting dynamic optimization, entitled JModelica. The JModelica platform relies on the established modeling language Modelica, and a recent extension, Optimica, that enables encoding of dynamic optimization problems based on Modelica models. The JModelica platform consists of software for translating high-level Modelica and Optimica source codes into C code which is then compiled and linked with numerical optimization solvers. The C model API is designed with flexibility in mind and offers rich opportunities for algorithm researchers to integrate new and innovative methods into the platform. Ultimately, the JModelica platform may then serve a bridge between the academic community and industry, while promoting both development and industrial use of state of the art optimization algorithms.

7 References

[1] Johan Åkesson. *Tools and Languages for Optimization of Large-Scale Systems*. PhD thesis, Department of Automatic Control, Lund University, Sweden, November 2007.

[2] Johan Åkesson, Torbjörn Ekman, and Görel Hedin. Implementation of a Modelica Compiler using JastAdd Attribute Grammars. *Science of Computer Programming*, 2009. Accepted for publication, to appear.

[3] Mats Andersson. *Object-Oriented Modeling and Simulation of Hybrid Systems*. PhD thesis, Department of Automatic Control, Lund Institute of Technology, Sweden, December 1994.

[4] J. Andreasson. Enhancing active safety by extending controllability—How much can be gained? In *Proceedings of IEEE Intelligent Vehicles Symposium*, Xi'an, Shaanxi, China, 2009. IEEE. Submitted for publication.

[5] Karl Johan Åström, Hilding Elmqvist, and Sven Erik Mattsson. Evolution of continuous-time modeling and simulation. In *Proceedings of the 12th European Simulation Multiconference, ESM'98*, pages 9–18, Manchester, UK, June 1998. Society for Computer Simulation International.

[6] B. M. Bell. CppAD Home Page, 2008. `http://www.coin-or.org/CppAD/`.

[7] R. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, N.J., 1957.

[8] John T. Betts. *Practical Methods for Optimal Control Using Nonlinear Programming*. Society for Industrial and Applied Mathematics, 2001.

[9] L.T. Biegler, A.M. Cervantes, and A Wächter. Advances in simultaneous strategies for dynamic optimization. *Chemical Engineering Science*, 57:575–593, 2002.

[10] Dynasim AB. Dynasim AB Home Page, 2008. `http://www.dynasim.se`.

[11] Torbjörn Ekman, Görel Hedin, and Eva Magnusson. JastAdd, 2008. http://jastadd.cs.lth.se/web/.

[12] Inc. Enthought. IPython FrontPage, 2009. `http://ipython.scipy.org/moin/`.

[13] Inc. Enthought. SciPy, 2009. `http://www.scipy.org/`.

[14] Python Software Foundation. ctypes: A foreign function library for Python, 2009. `http://docs.python.org/library/ctypes.html`.

[15] Python Software Foundation. Python Programming Language – Official Website, 2009. `http://www.`

python.org/.

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

[17] Staffan Haugwitz, Johan Åkesson, and Per Hagander. Dynamic start-up optimization of a plate reactor with uncertainties. *Journal of Process Control*, 2009. doi:10.1016/j.jprocont.2008.07.005.

[18] Görel Hedin, Johan Åkesson, and Torbjörn Ekman. Building DSLs by leveraging base compilers—from Modelica to Optimica. *IEEE Software*. Submitted for publication.

[19] Görel Hedin and Eva Magnusson. JastAdd: an aspect-oriented compiler construction system. *Science of Computer Programming*, 47(1):37–58, 2003.

[20] J. Hunter, D. Dale, and M. Droettboom. matplotlib: python plotting, 2009. http://matplotlib.sourceforge.net/.

[21] IEEE. Standard VHDL Analog and Mixed-Singnal Extensions. Technical report, IEEE, 1997.

[22] Center for Applied Scientific Computing Lawrence Livermore National Laboratory. SUNDIALS (SUite of Nonlinear and DIfferential/ALgebraic equation Solvers), 2009. https://computation.llnl.gov/casc/sundials/main.html.

[23] D.B. Leineweber, I. Bauer, H.G. Bock, and J.P. Schloder. An efficient multiple shooting based reduced SQP strategy for large-scale dynamic process optimization. Part 1: theoretical aspects. *Computers and Chemical Engineering*, 27(2):157–166, 2003.

[24] D.B. Leineweber, A. Schafer, H.G. Bock, and J.P. Schloder. An efficient multiple shooting based reduced SQP strategy for large-scale dynamic process optimization - Part II: Software aspects and applications. *Computers and Chemical Engineering*, 27(2):167–174, 2003.

[25] T. Oliphant. Numpy Home Page, 2009. http://numpy.scipy.org/.

[26] L. S. Pontryagin, V. G. Boltyanskii, R. V. Gamkrelidze, and E. F. Mishchenko. *The Mathematical Theory of Optimal Processes*. John Wiley & Sons Inc, 1962.

[27] Process Systems Enterprise. gPROMS Home Page, 2007. http://www.psenterprise.com/gproms/index.html.

[28] The Modelica Association. The Modelica Association Home Page, 2007. http://www.modelica.org.

[29] Triple-J Group for Molecular Cell Physiology University of Stellenbosch. PySUNDIALS: Python SUite of Nonlinear and DIfferential/ALgebraic equation Solvers, 2009. http://pysundials.sourceforge.net/.

[30] V. Vassiliadis. *Computational solution of dynamic optimization problem with general differnetial-algebraic constraints*. PhD thesis, Imperial College, London, UK, 1993.

[31] Andreas Wächter and Lorenz T. Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical Programming*, 106(1):25–58, 2006.