Modeling and Optimization with Modelica and Optimica Using the JModelica.org Open Source Platform

Johan Åkesson[a,b] Tove Bergdahl[a] Magnus Gäfvert[a] Hubertus Tummescheit[a]
[a] Modelon AB, Sweden
[b] Department of Automatic Control, Lund University, Sweden

Abstract

This paper reports a new Modelica-based open source project entitled JModelica.org, 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.org 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. In this contribution, an overview of the platform is presented and the main features of JModelica.org are highlighted.

Keywords: Modelica; Optimica; Optimization; Model Predictive Control

1 Introduction

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.

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 [30] and Modelica [44]. 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.

This contribution reports a new Modelica-based open source initiative targeted at dynamic optimization entitled JModelica.org. JModelica.org [36] is a novel open source project with the mission:

"To offer a community-based, free, open source, accessible, user and application oriented Modelica environment for optimization and simulation of complex dynamic systems, built on well-recognized technology and supporting major platforms."

JModelica.org is primarily focused on dynamic optimization of Modelica models. To meet this end, JModelica.org supports Optimica, which is an extension to the Modelica language that offers language constructs for encoding of cost functions, constraints and the optimization interval with fixed or free end points. The platform consists of compilers for translating Modelica and Optimica models into C and XML code, a C API for evaluation of model equations and Python bindings to enable scripting and custom algorithm development. The software is distributed freely under the GPL license.

The paper is outlined as follows. In Section 2, a review of optimization tools and the Optimica extension are given. Section 3 describes the JModelica.org platform. Previous case studies performed based on JModelica.org, and the opportunities provided by abstract syntax tree access are discussed in Section 4. In Section 5, an example of a model predictive control application is given. The paper ends with a summary and comments on future work in Section 6.

2 Background

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, [4], VHDL-AMS [30], and Modelica [44]. See [5] for a comprehensive overview of the evolution of continuous-time simulation languages and tools.

2.1 Optimization Tools

There are several tools for optimization on the market, offering different features. In essence, three different categories of tools can be distinguished, although the functionality is sometimes overlapping. *Model integration tools* addresses the problem of interfacing several design tools into a a single computation environment, where analysis, simulation and optimization can be performed. Examples are ModelCenter, [41], OptiY, [40], modeFRONTIER [21], and iSIGHT, [11]. Typically, such tools are dedicated to design optimization of extremely complex systems which may be composed from subsystems encoded in different tools. Accordingly, model integration tools typically offers interfaces to CAD and finite element software as well as simulation tools for, e.g., mechanical and hydraulic systems. As a result of the heterogeneity and complexity of the target models, models are usually treated as black boxes, i.e. the result of a computation is propagated to the tool, but the structure of a particular model is not explored. Accordingly, heuristic optimization algorithms which do not require derivative information or detailed structural information, are usually employed. In addition, model integration tools often have sophisticated features supporting model approximation and visualization.

Several *Simulation tools* comes with optimization add-ons, e.g., Dymola [14], gPROMS [42] and Jacobian [37]. Such tools typically offer strong support for modeling of physical systems and simulation. The level of support for optimization in this category differs between different tools. Dymola, for example, offers add-ons for parameter identification and design optimization, [18]. gPROMS, on the other hand, also offers support for solution of optimal control problems. Tools in this category are usually limited to a predefined set of optimization algorithms. Integration of new algorithms may be difficult if the tools do not provide the necessary API:s.

In the third category we have *numerical packages* for dynamic optimization, often developed as part of research programs. Examples are ACADO [39], Muscod II [46], and DynoPC [33], which is based on Ipopt [48]. Such packages are typically focused on efficient implementation of an optimization algorithm for a particular class of dynamic systems. Also, detailed information about the model to optimize is generally required in order for such algorithms to work, including accurate derivatives and in some cases also sparsity patterns. Some of the packages in this category are also targeting optimal control and estimation problems in real-time, e.g., non-linear model predictive control, which require fast convergence. While these packages offer state of the art algorithms, they typically come with simple or no user interface. Their usage is therefore limited due to the effort required to code the model and optimization descriptions.

The JModelica.org platform is positioned to fill the gap left between simulation tools offering optimization capabilities and state of the art numerical algorithms. Primarily, target algorithms are gradient based methods offering fast convergence. Never the less, JModelica.org is well suited for use also with heuristic direct search methods; the requirements with respect to execution interface is typically a subset of the requirements for gradient based methods. The problems addressed by model integration tools is currently beyond the scope of JModelica.org, even though its in-

```
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
   ⑧(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.

tegration with Python offers extensive possibilities to develop custom applications based on the solution of simulation and optimization problems.

2.2 Optimica

The Optimica extension is discussed in detail [1, 2]. 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

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

and

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

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.

3 The JModelica.org platform

In order to demonstrate the feasibility and effectiveness of the proposed Optimica extension, a prototype

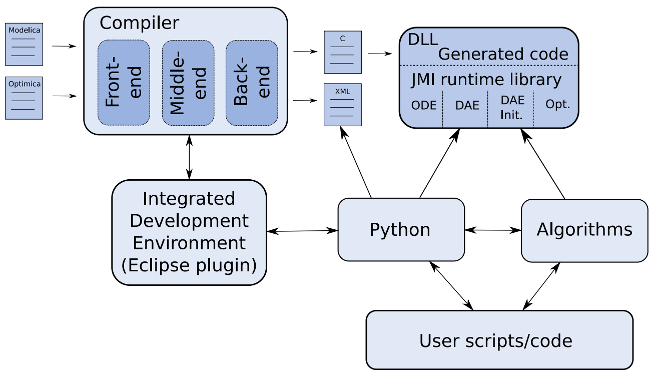

Figure 1: Overview of the JModelica.org platform architecture.

```
model M
  Real x;
equation
  x = 1;
end M;
```

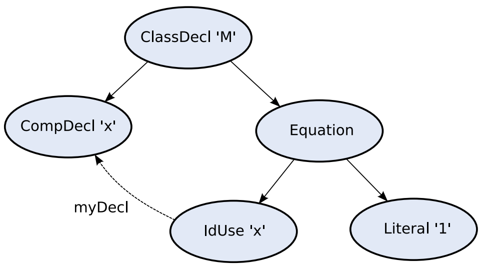

Figure 2: A simple Modelica model (left) and its corresponding abstract syntax tree (right). The dashed arrow represents the reference attribute `myDecl` which binds an identifier to its declaration.

compiler was developed, [1]. Currently, the initial prototype compiler is being developed with the objective of creating a Modelica-based open source platform focused on dynamic optimization.

The architecture of the JModelica.org platform is illustrated in Figure 1. The platform consists essentially of two main parts: the compiler and the JModelica.org Model Interface (JMI) runtime library. The compiler transforms Modelica and Optimica source code into a flat model description and then generates C and XML code. The generated C code contains the actual model equations in a format suitable for efficient evaluation, whereas the XML code contains model meta data, such as variable names and parameter values. The JMI runtime library provides a C interface which in turn can be interfaced with numerical algorithms. There is also an Eclipse plug-in and a Python integration module under development. In this section, the key parts of the JModelica.org platform will be described.

3.1 Compiler Development—JastAdd

Compiler construction has traditionally been associated with intricate programming techniques within the area of computer science. Recent research effort has, however, resulted in new compiler construction frameworks that are easier to use and where it is feasible to develop compilers with a comparatively reasonable effort. One such framework is JastAdd [28, 17]. JastAdd is a Java-based compiler construction framework based on concepts such as object-orientation, aspect-orientation and reference attributed grammars [15]. At the core of JastAdd is an abstract syntax specification, which defines the structure of a computer program. Based on an abstract syntax tree (AST), the compiler performs tasks such as *name analysis*, i.e, finding declarations corresponding to identifiers, *type analysis*, i.e., verifying the type correctness of a program, and code generation.

The JastAdd way of building compilers involves specification of *attributes* and *equations* based on the abstract syntax specification. This feature is very similar to ordinary Knuth-style attribute grammars [32] but enhanced with reference attributes. Accordingly, attributes may be used to specify, declaratively, links between different nodes in the AST. For example, identifier nodes can be bound to their declaration nodes. In Figure 2, an example of a small Modelica program and its corresponding AST is shown. Notice how the reference attribute `myDecl` links an identifier (`IdUse`) to its declaration (`CompDecl`).

JastAdd attributes and equations are organized into separate *aspects*, which form a convenient abstraction for encoding of cross cutting behavior. Typically, implementation of a semantic function, for example type analysis, involves adding code to large number of classes in the AST specification. Using aspects, much like in AspectJ [31], cross cutting behavior can be modularized in a natural way. In addition, this approach is the basis for one of the distinguishing features of JastAdd: it enables development of modularly extensible compilers. This means that it is feasible to develop, with a comparatively moderate effort, modular extensions of an existing JastAdd compiler without changing the core compiler. This feature has been used in the implementation of the JModelica.org Modelica and Optimica compilers, where the Optimica compiler is a fully modular extension of the core Modelica compiler.

The JastAdd compiler transforms the JastAdd specification into pure Java code, where the definition of the abstract grammar translates into Java classes corresponding to Modelica classes, components, functions, and equations. The JastAdd attributes are woven into the Java classes as methods. In addition, methods for traversing an AST and query properties of a particular AST class, e.g., obtain a list of variables contained in a class declaration, are automatically generated. As a result of this approach, compilers produced by JastAdd are in the form of standard Java packages, which in turn can be integrated in other applications. It is therefore not necessary to know the particular details of how to write JastAdd specifications in order to use the JModelica.org compilers, knowledge of Java is generally sufficient.

3.2 The Modelica and Optimica Compilers

At the core of the JModelica.org platform is a Modelica compiler that is capable of transforming Modelica code into a flat representation and of generating C code. In the Modelica compiler, several design strategies, for example name look-up, developed for a Java compiler developed using JastAdd [16], were reused. For additional details on the implementation of the compiler, see [3].

In order to support also the Optimica extension, a modular extension of the core Modelica compiler has been developed. The extended compiler is capable of translating standard Modelica enhanced with the new Optimica syntax presented in Section 2.2. The Optimica extension is reported in more detail in [27].

The JModelica.org Modelica compiler currently supports a subset of Modelica version 3.0. The Modelica Standard Library version 3.0.1 can be parsed and the corresponding source AST can be constructed. Flattening support is more limited, but is being continuously improved.

3.3 Code Generation

The JModelica.org offers a code generation framework implemented in Java as part of the compilers. The framework facilitates development of custom code generation modules and is based on *templates* and *tags*. A template is used to specify the structure of the generated code and tags are used to define elements of the template which is to be replaced by generated code. In order to develop a custom code generation module, the user needs to define a template and a set of tags, and then implement the actual code generation behavior corresponding to each tag. In order to perform the latter, the AST for the flattened Modelica model is typically used, where objects corresponding to declarations, equations and functions are queried for information used to generate the target code.

The JModelica.org platform contains two code generation modules, one for C and one for XML. The generated C code contains the model equations and is intended to be compiled and linked with the JModelica.org Model Interface (see below) in order to offer efficient evaluation of the model equations. The XML output consists of model meta data such as specifications of variables, including their names, attributes and type. Also, the XML output includes a separate file for parameter values. The XML output is similar to what is discussed within the FMI initiative [12], and the intention is for the JModelica.org XML output to be compliant with FMI once finalized. In addition, there is on-going work aimed to develop an XML specification for flattened Modelica models, including variable declarations, functions, and equations [9]. The objective is for JModelica.org to be compliant also with this specification.

3.4 C API

The JModelica.org platform offers a C API, entitled the JModelica.org Model Interface (JMI[1]), suitable for integration with numerical algorithms. The interface provides functions for accessing and setting parameter and state values, for evaluation of the DAE residual function and for evaluation of cost functions and constraints specified in an Optimica model. In addition, Jacobians and sparsity patterns can be obtained for all functions in the interface. To meet this end, a package for automatic differentiation, CppAD [6], has been integrated into JMI. The JMI code is intended to be compiled with the C code that is generated by the compiler into an executable, or into a shared object file.

The JMI interface consists of four parts: the ODE interface, the DAE interface, the DAE initialization interface, and the Optimization interface. These interfaces provide access to functions relevant for different parts of the optimization specification. The ODE and DAE interfaces provide evaluation functions for the right hand side of the ODE and the DAE residual function respectively. The DAE initialization problem provides functions for solving the DAE initialization problem, whereas the Optimization interface provides functions for evaluation of the cost functions and the constraints.

[1]Notice that this acronym is unrelated to Java Metadata interface

3.5 Interactive Environment—Python

Solution of engineering problems typically involves atomization of tasks in the form of user scripts. Common examples are batch simulations, parameter sweeps, post processing of simulation results and plotting. Given that JModelica.org is directed towards scientific computing, Python, see [23], is an attractive option. Python is a free open-source highly efficient and mature scripting language with strong support in the scientific community. Packages such as NumPy [38] and SciPy [20], and bindings to state-of-the art numerical codes implemented in C and Fortran make Python a convenient glue between JModelica.org and numerical algorithms. In addition, IPython [19] with the visualization package matplotlib [29] and the PyLab mode offer an interactive numerical environment similar to the ones offered by Matlab and Scilab.

The JModelica.org Pyhon package includes sub packages for running the compilers, for managing file input/output of simulation/optimization results and for accessing the function provided by the JMI interface. The compilers are run in a Java Virtual Machine (JVM) which is connected to the Python environment by the package JPype, [35]. One of JPype's main features is to enable direct access to Java objects from a Python shell or script. This feature is used to communicate with the compilers, but can also be used to retrieve the ASTs generated by the compilers. The latter feature enables the user to traverse and query the ASTs interactively, see 4.2 for a discussion on example usages of this feature.

The integration of the JMI is based on the ctypes package [22]. Using ctypes, a dynamically linked library (DLL) can be loaded into Python, All the contained functions of the DLL are then exposed and can be called directly from the Python shell. In order to enable use of Numpy arrays and matrices as arguments to the JMI functions, the argument types has been explicitly encoded using standard features of ctypes. In order to provide a more convenient interface to the JMI functions, a Python class, `Model` has been created. This class encapsulates loading of a DLL and typing of the JMI functions, and also provides wrapper functions supporting Python exceptions. In addition, upon creation of a `Model` class, the generated XML meta data files are loaded and parameter values and start attributes are set in the loaded model instance. `Model` objects can then be manipulated, e.g., by setting new parameter values, or passed as an argument to a simulation or optimization algorithm.

3.6 Optimization Algorithms

The JModelica.org platform offers two different algorithms for solving dynamic optimization problems. The first is a simultaneous optimization method based on orthogonal collocation on finite elements [7]. Using this method, state and input profiles are parameterized by Lagrange polynomials which are based on Radau points. This method corresponds to a fully implicit Runge-Kutta method, and accordingly it possesses well known and strong stability properties. By parameterizing the variable profiles by polynomials, the dynamic optimization problem is translated into a non-linear programming (NLP) problem which may be solved by a numerical NLP solver. This NLP is, however, very large. In order to efficiently find a solution to the NLP, derivative information as well as the sparsity patterns of the constraint Jacobians need to be provided to the solver. The simultaneous optimization algorithm has been interfaced with the large-scale NLP solver Ipopt [48], which has been developed particularly to solved NLP problems arising in simultaneous dynamic optimization methods. The algorithm is implemented in C as an extension of JMI, and provides an example of how to implement algorithms based on the JMI functions. In particular, Jacobians computed by CppAD is used, including sparsity patterns.

In addition to the simultaneous optimization algorithm, JModelica.org contains a multiple shooting algorithm, [8]. The algorithm is based on an integrator which is used to simulate the system dynamics and thereby evaluate the cost function, and an optimization algorithm which modifies the optimization variables. Typically, the optimization variables are Modelica parameters in the case of a design or parameter optimization problem, or parameters resulting from discretization of a control input. The multiple shooting algorithm is implemented in Python, and relies on the integrator SUNDIALS [34], its Python interface PySUNDIALS [47], and the optimization algorithm scipy_slsqp, which is included in Scipy. In order to improve the convergence of the optimization algorithm *sensitivities* are computed and propagated to the optimization algorithm. The sensitivities are computed using SUNDIALS. The implementation serves also as an example of how to develop algorithms based on the JModelica.org Python interface. The multiple shooting algorithm is described in more detail in [43].

The above algorithms are both based on the availability of derivatives. For some optimization problems, it is not possible to reliably compute derivatives, and accordingly, numerical optimization algorithms requiring derivative information may fail. This situation may occur for certain classes of hybrid systems. In such cases, heuristic methods which do not require derivative methods may be better suited. Examples of such methods are genetic algorithms, pattern matching, simulated annealing, and simplex (Nelder-Mead). Some methods of this class are freely available for Python, see the OpenOpt project [13] for more information, and may be integrated with the JMI Python interface.

4 Applications

4.1 Dynamic optimization

Prototype versions of the JModelica.org software has been used successfully in applications in different domains. In [26], an application is reported where optimal start-up trajectories are computed for a plate reactor. The problem is challenging not only due to the complexity of the model (approx. 130 differential and algebraic equations), but also due to non-linear and in some conditions unstable dynamics. A main challenge in this project was to obtain trajectories robust to parameter variations. The procedure of finding acceptable solutions was highly iterative in that the cost function and the constraints required several redesigns. The high-level specification framework in combination with automatic code generation offered by Optimica and the JModelica.org platform proved very useful in this respect.

The prototype software has as also been used in a number of other projects involving vehicle systems. For example, in [10] optimal tracks for racing cars were optimized, and in [45], optimal rail profiles were computed for a novel mass transportation system, the NoWait train concept. Other examples where Optimica has been used are reported in [25] where minimum time optimization for an industrial robot is considered and in [24] where an optimal control application of a pendulum system is reported.

4.2 Using ASTs

As described above, the JModelica.org compilers provide direct access to abstract syntax trees (ASTs). The ASTs are abstract representations of Modelica models, and provides a means to access the contents of a Model in a structured and programmatic way. Three different types of ASTs are used during the procedure of producing a flat Modelica representation: the *source* AST, the *instance* AST, and the *flat* AST. The ASTs in the JModelica.org org compilers consists of standard Java objects instantiated from the AST classes produced by the JastAdd compilers. Also, means to traverse the AST are provided automatically, in addition to the methods corresponding to attributes defined in the compiler.

The source AST results from parsing of a Modelica source file. Its structure corresponds precisely to the actual structure of the code, but the details of the concrete syntax has been removed. Given the source AST, a number of queries can be performed. For example, the AST may be traversed and for each class declaration, the documentation annotation and the signatures of the public parameters may be extracted and pretty printed according to a HTML template. Another example would be to traverse the AST and output the default values of all parameters in XML format.

Based on a particular class declaration in the source AST, an instance AST may be constructed. The instance AST differs from the source AST in that in the instance AST the components structurally contained in a component declaration is explicitly represented. Also, in the instance tree, modifications, e.g., class and component redeclarations take effect. In fact, the key to constructing the instance tree is to handle *modification environments* consisting of an ordered set of modifications applicable to a particular class or component instance. Construction of the instance AST is described in [3]. Access to the instance AST enables several analyses to be performed. For example, the instance AST may be traversed, and for each primitive variable encountered the corresponding modification environment may be retrieved and output to file.

The flat AST, corresponding to a flattened Modelica model, is constructed by traversing the instance AST and collecting all primitive variables, equations, algorithms and functions. The flat AST offers a predefined API for retrieving variables of the primitive types. For example, Java methods are provided for retrieving all parameters of Real type, for retrieving all differentiated variables, for querying if a variable occurs linearly in the model equations etc. The flat AST is typically used as a basis for code generation.

5 An Example

We consider the Continuously Stirred Tank Reactor (CSTR) process depicted in Figure 3. The temperature and the reactant concentration of the inlet flow are constant and the control input of the process, u corresponds to the coolant flow. The coolant temperature,

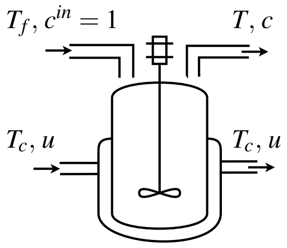

Figure 3: A schematic figure of the CSTR process.

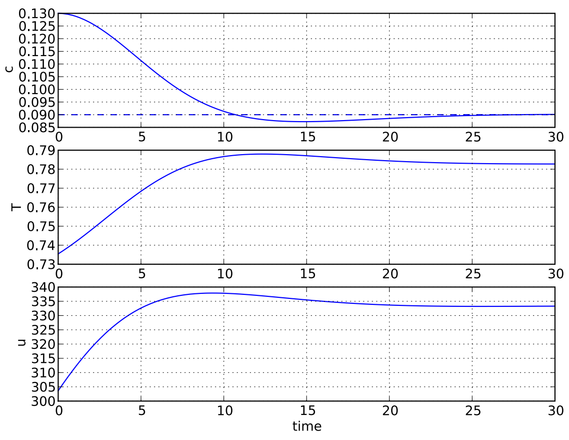

Figure 4: Optimization result for the CSTR example.

T_c, is constant. A dynamic model for an exothermic reaction is then given by

$$\begin{aligned}\dot{c} &= \beta(1-c) - ke^{-N/T}c \\ \dot{T} &= \beta(T_f - T) + ke^{-N/T}c - \alpha(T - T_c)u\end{aligned} \tag{4}$$

where c is the normalized concentration in the reactor, T is the normalized reactor temperature, and β, k, N, and α are physical parameters for the process.

Based on the CSTR model, the following dynamic optimization problem can be formulated:

$$\min_{u(t)} \int_0^{t_f} q_1(c_r - c)^2 + q_2(T_r - T)^2 + r(u_r - u)^2 dt \tag{5}$$

subject to the dynamics (4). The cost function penalizes deviations from a desired operating point given by target values c_r, T_r and u_r for c, T and u respectively. Starting at fixed initial conditions, the optimal solution transfers the system from one operating point to another.

In this case, the numerical solver IPOPT [48] is used to solve the transcribed NLP resulting from direct collocation. The result of the optimization is shown in Figure 4.

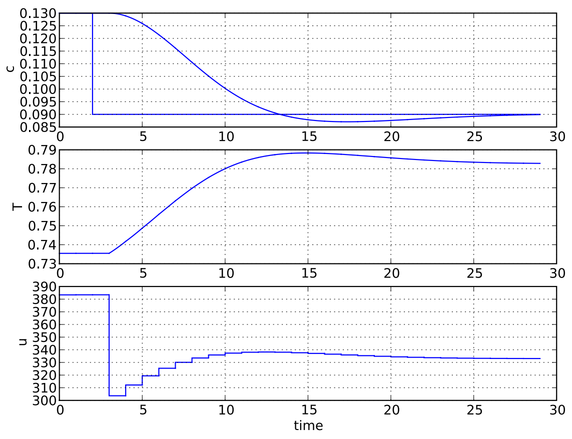

Figure 5: Simulation result of the MPC.

The optimal control problem formulated above can also be used in conjunction with other algorithms available in Scipy. To demonstrate this, a simple model predictive controller (MPC) has been implemented. The MPC control strategy is based on the receding horizon principle, where an open loop optimal control problem is solved in each sample. Simulation of an MPC requires joint simulation of the plant and solution of the optimal control problem. Such operations are easily encoded in Python. The result of executing the MPC is shown in Figure 5.

6 Summary and future work

In this paper, the JModelica.org open source platform has been presented. The platform features compilers written in JastAdd/Java, Optimica support, a C model API, and XML export. The compilers and the C API for evaluation of the model equations have been interfaced with Python, in order to provide an environment for scripting and development of custom applications. In addition, the abstract syntax trees representing the Modelica source code, the instance hierarchy and the flattened model are accessible.

Future plans include improvement of the Modelica compliance of the compiler front-end, integration of additional numerical optimization algorithms, simulation support, and support for heuristic optimization methods such as simulated annealing and genetic algorithms. Also, an Eclipse plug-in is under development where current research on custom IDE development based on JastAdd is explored.

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. Optimica—an extension of modelica supporting dynamic optimization. In *In 6th International Modelica Conference 2008*. Modelica Association, March 2008.

[3] Johan Åkesson, Torbjörn Ekman, and Görel Hedin. Implementation of a modelica compiler using jastadd attribute grammars. *Science of Computer Programming*, July 2009. doi:10.1016/j.scico.2009.07.003.

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

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

[8] H.G. Bock and K. J. Plitt. A multiple shooting algorithm for direct solution of optimal control problems. In *Ninth IFAC world congress*, Budapest, 1984.

[9] F. Casella, D. Filippo, and J. Åkesson. n XML Representation of DAE Systems Obtained from Modelica Models. In *In 7th International Modelica Conference 2009*. Modelica Association, 2009.

[10] Henrik Danielsson. Vehicle path optimisation. Master's Thesis ISRN LUTFD2/TFRT--5797--SE, Department of Automatic Control, Lund University, Sweden, June 2007.

[11] Dassault Systèmes. iSIGHT Home Page, 2009. `http://www.simulia.com/products/isight.html`.

[12] DLR, Dynasim, ITI and QTronic. The functional model interface. Draft.

[13] Dmitrey L. Kroshko. OpenOpt Home Page, 2009. `http://openopt.org/Welcome`.

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

[15] T. Ekman and G. Hedin. Rewritable Reference Attributed Grammars. In *Proceedings of ECOOP 2004*, volume 3086 of *LNCS*, pages 144–169. Springer-Verlag, 2004.

[16] Torbjön Ekman and Görel Hedin. The jastadd extensible java compiler. In *Proceedings of OOPSLA 2007*, 2007.

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

[18] H. Elmqvist, H. Olsson, S.E. Mattsson, D. Brück, C. Schweiger, D. Joos, and M. Otter. Optimization for design and parameter estimation. In *In 7th International Modelica Conference 2009*. Modelica Association, 2005.

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

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

[21] ESTECO. modeFRONTIER Home Page, 2009. `http://www.esteco.com/`.

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

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

[24] P. Giselsson, J. Åkesson, and A. Robertsson. Optimization of a pendulum system using optimica and modelica. In *In 7th International Modelica Conference 2009*. Modelica Association, 2009.

[25] M. Hast, J. Åkesson, and A. Robertsson. Optimal Robot Control using Modelica and Optimica. In *In 7th International Modelica Conference 2009*. Modelica Association, 2009.

[26] 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.

[27] Görel Hedin, Johan Åkesson, and Torbjön Ekman. Building DSLs by leveraging base compilers—from Modelica to Optimica. *IEEE Software*, 2009. Submitted for publication.

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

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

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

[31] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *LNCS*, 2072:327–355, 2001.

[32] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, June 1968. Correction: *Mathematical Systems Theory* 5, 1, pp. 95-96 (March 1971).

[33] Y.D. Lang and L.T. Biegler. A software environment for simultaneous dynamic optimization. *Computers and Chemical Engineering*, 31(8):931–942, 2007.

[34] 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.

[35] S. Menard. JPype Home Page, 2009. http://jpype.sourceforge.net/.

[36] Modelon AB. JModelica Home Page, 2009. http://www.jmodelica.org.

[37] Numerica Technology. Jacobian Home Page, 2009. http://www.numericatech.com/jacobian.htm.

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

[39] OPTEC K.U. Leuven. ACADO Home Page, 2009. http://www.acadotoolkit.org/.

[40] OptiY. OptiY Home Page, 2009. http://www.optiy.de/.

[41] Phoenix Integration. ModelCenter Home Page, 2009. http://www.phoenix-int.com/software/phx_modelcenter.php.

[42] Process Systems Enterprise. gPROMS Home Page, 2009. http://www.psenterprise.com/gproms/index.html.

[43] J. Rantil, J. Åkesson, C. Führer, and M. Gäfvert. Multiple-Shooting Optimization using the JModelica.org Platform. In *In 7th International Modelica Conference 2009*. Modelica Association, 2009.

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

[45] Jan Tuszynskia, Mathias Persson, Johan Åkesson, Johan Andreasson, and Magnus Gäfvert. Model-based approach for design and validation of a novel concept of public mass transport. In *21st International Symposium on Dynamics of Vehicles on Roads and Tracks*, 2009. Accepted for publication.

[46] University of Heidelberg. MUSCOD-II Home Page, 2009. http://www.iwr.uni-heidelberg.de/~agbock/RESEARCH/muscod.php.

[47] 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/.

[48] 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.