Event support for simulation and sensitivity analysis in CasADi for use with Modelica and FMI

Joel Andersson[1] James Goppert[2]

[1]Freelance software developer and consultant, USA, joel@jaeandersson.com
[2]School of Aeronautics and Astronautics, Purdue University, USA, jgoppert@purdue.edu

Abstract

CasADi is an open-source framework that can be used to efficiently solve optimization problems involving user-defined ODE/DAE models. Supported solution methods include so-called shooting methods, where solvers for initial-value problems in ODEs or DAEs are referenced inside in nonlinear programming (NLP) formulations. In order to solve such NLP formulations with gradient-based algorithms, CasADi implements a fully automatic sensitivity analysis. This analysis includes forward sensitivity analysis, adjoint sensitivity analysis as well as the calculation of higher-order sensitivities for the ODE/DAE models. Because of the variational (differentiate-then-integrate) approach used, the numerical solution can be performed with variable-step size, variable-order integrators such as those from the SUNDIALS suite.

In this work, we present a generalization of the sensitivity analysis support in CasADi to systems with events, as are common in real-world cyber-physical models. In particular, the event extension enables us to formulate and solve optimization problems with such event systems, without a priori knowledge of the number and ordering of events. Ultimately, we expect the proposed approach to be compatible with general cyber-physical models formulated in Modelica or available as model-exchange FMUs.

We demonstrate the proposed approach for two proof-of-concept examples; the classical bouncing ball written in CasADi directly and a simple hybrid DAE describing a breaking spring formulated in Modelica and imported symbolically into CasADi. In the examples, we show that the forward sensitivities calculated to high precision using the proposed approach are consistent with a cruder finite-difference approximation and provide an example of how they can be embedded into optimization formulations. We discuss how the approach can be extended to handle standard FMUs, adhering to FMI 2 or FMI 3, as well as non-trivial Modelica models imported via a symbolic interface based on the emerging Base Modelica standard.

Keywords: Hybrid DAEs, sensitivity analysis, CasADi, Modelica, FMI

1 Introduction

Dynamic models describing cyber-physical systems often include events that are triggered when some conditions are met. These events can arise both from the need to faithfully capture the physics, e.g. an object transitioning from being stationary to starting to slide, or to capture the modes in control systems. Physical modeling environments, such as those based on Modelica, allow events to be efficiently described and transformed into a canonical form compatible with numerical solvers. For Modelica, the corresponding form is a *hybrid differential-algebraic equation* (DAE) as described by the language specification (Modelica Association 2021). For a hybrid DAE in a standard form, events are generally triggered by zero-crossing conditions for a set of event indicators, which are evaluated along with the DAE. Certain numerical solvers such as those from the SUNDIALS suite (Hindmarsh et al. 2005) used in this work, are able to monitor the event indicators for zero-crossings and stop the integration prematurely if an event is detected. At the detected event, the system is then updated according to the finite state machine semantics described in the hybrid DAE representation before the DAE integration is resumed.

1.1 Events in dynamic optimization

The handling of events using zero-crossing events and event transitions is the standard approach for hybrid DAE simulation. For dynamic optimization problems, i.e. optimization problems where the hybrid DAE enters as constraints in the formulation, the standard approach is instead to partition the time horizon into multiple *stages* (or *phases*) with events happening between the different stages but not within them. The time durations between events then become additional decision variables of the optimization problem. To illustrate, if we have a single event at (a priori unknown) time T, the physical time variable t is substituted in the first stage with a dimensionless time τ according to $t = T\,\tau$. We can then proceed to solve the optimization problem as if the event times were known with stage durations added as an additional optimization variables. While this approach has proven useful in numerous applications, it is not as general as the hybrid DAE representation used for simulation. In particular, it requires a priori knowledge of the number of stages, which is often not available.

Using the approach proposed here, this reformulation to a multi-stage problem, with associated restrictions, is no longer needed. Instead, we are able to embed the hybrid

DAEs directly in the dynamic optimization formulations and still get the exact first and second order sensitivity information needed by gradient-based numerical optimization methods.

1.2 CasADi

CasADi (J. A. E. Andersson et al. 2019) is an open-source software package for C++, Python, MATLAB and Octave. It offers versatile environent that in particular can be used to solve a range of different numerical optimization problems, using different methods and solvers. In particular, CasADi can be used to efficiently solve numerical optimal control problems, i.e. optimization problem constrained by differential equations. At the core of CasADi is a symbolic framework implementing *algorithmic differentiation* (AD) in both forward and reverse (adjoint) modes. In addition to AD, symbolic expressions can be used for efficient evaluation, either in virtual machines or in generated, self-contained C code. Importantly, the symbolic expressions can embed calls to user-defined, differentiable *function objects*. Such function objects can be defined in number of different ways, including from other symbolic expressions, by linear or nonlinear systems of equations or user defined code. In (Joel Andersson 2023), it was shown how differentiable CasADi function objects could be created from *functional mock-up units* (FMUs) adhering to the *functional mock-up interface* (FMI) standard. In this work, we present an extention of another imporant type of function objects in CasADi, *Integrator* instances, which are used to simulate and perform sensitivity analysis for differential equations. A relatively comprehensive and up-to-date description of this functionality is presented in Section 2. In Section 3, we will show how the integrators were extended to support general events handling, while still retaining efficient and accurate differentiability.

1.3 Related work

Sensitivity analysis and numerical optimization for hybrid dynamic systems have been performed previously, in particular in the Julia environment, using integrators formulated in the *DifferentialEquations.jl* package (Rackauckas and Nie 2017). For models available as expressions, derivatives can be calculated analytically by differentiating the entire algorithm, giving a integrate-then-differentiate approach. It is our understanding that this approach, unlike the variational approach presented here, can not be readily used with models formulated in Modelica or provided as FMUs. We refer to (Corner, C. Sandu, and A. Sandu 2019) for a recent overview of methods for hybrid sensitivity analysis.

2 Simulation and sensitivity analysis in CasADi

CasADi can be used to solve initial value problems (IVP) in ordinary differential equations (ODEs) or differential-algebraic equations (DAEs) with fully automatic sensitivity analysis. This support, which has existed since early versions of CasADi, has been extended and improved over the years. In the following, we provide a description of the current algorithm, which largely corresponds to the refactoring of the functionality which enabled the use of FMI models, as described in (Joel Andersson 2023). In Section 3, we will show how this formulation can be extended to support events, while still maintaining efficient, analytic differentiability.

The dynamic systems considered, as of CasADi 3.6, are semi-explicit DAEs with quadratures:

$$\begin{cases} \dot{x}(t) = f_{\text{ode}}(t, x(t), z(t), p, u(t)) \\ 0 = f_{\text{alg}}(t, x(t), z(t), p, u(t)) \\ \dot{q}(t) = f_{\text{quad}}(t, x(t), z(t), p, u(t)), \end{cases} \tag{1}$$

where $t \in \mathbb{R}$ is time (or some other independent variable), $x(\cdot) \in \mathbb{R}^{n_x}$ is a state vector, $z(\cdot) \in \mathbb{R}^{n_z}$ is a vector of algebraic variables, $q(\cdot) \in \mathbb{R}^{n_q}$ is a state vector that does not appear in the right-hand-side, $p \in \mathbb{R}^{n_p}$ is a (tunable) parameter and $u(\cdot) \in \mathbb{R}^{n_u}$ is a control input, which is assumed to be piecewise constant. If piecewise constant control inputs are too restrictive for a particular application, piecewise polynomial approximations can be handled by adding additional state variables (e.g. defined by $\dot{x}_{\text{piecewise linear}} = u_{\text{piecewise constant}}$). The quadrature states in this context are especially important for calculating integral terms but are also used in adjoint sensitivity analysis.

We assume that any DAE is of index-1, i.e. in particular that the Jacobian of f_{alg} with respect to z exists and is invertible. For ODEs, z and $f_{\text{alg}}(\cdot)$ have dimension zero. If the index-1 assumption does not hold, an index reduction should be performed prior to simulation, which has been implemented both in CasADi natively (for models given as symbolic expressions) and in coupled modeling environents, such as those based on Modelica.

To solve IVPs, the CasADi user creates *Integrator* instances. These are formed from a given the initial time t_0, an output time grid $[t_1, \dots, t_N]$ as well as symbolic expressions of the form (1), or more generally, a (differentiable) CasADi function object that calculates f_{ode}, f_{alg} and f_{quad} from given values for t, x, z, p and u:

$$\begin{aligned} &f : \mathbb{R} \times \mathbb{R}^{n_x} \times \mathbb{R}^{n_z} \times \mathbb{R}^{n_p} \times \mathbb{R}^{n_u} \to \mathbb{R}^{n_x} \times \mathbb{R}^{n_z} \times \mathbb{R}^{n_q} \\ &(t, x, z, p, u) \mapsto (f_{\text{ode}}, f_{\text{alg}}, f_{\text{quad}}) \end{aligned} \tag{2}$$

For the typical usage, an Integrator instance is a (differentiable) CasADi function object that given $x(t_0)$, p, the $u(t)$ trajectory and a *guess* for $z(t_0)$, calculates $x(t_k)$, $z(t_k)$ and $q(t_k)$ at all output times, $k = 1, \dots, N$. If the DAE has quadratures, $q(t_0)$ is assumed zero. We can write the function object defined by the integrator instance as follows:

$$\begin{aligned} &F : \mathbb{R}^{n_x} \times \mathbb{R}^{n_z} \times \mathbb{R}^{n_p} \times \mathbb{R}^{n_u \times N} \to \mathbb{R}^{n_x \times N} \times \mathbb{R}^{n_z \times N} \times \mathbb{R}^{n_q \times N} \\ &(x_0, z_0, p, u) \mapsto (x, z, q) \end{aligned} \tag{3}$$

Solving the initial value problem

The actual calculation of (3) takes places in the CasADi Integrator class, which relies on successive calls to one of the solver *plugins*. The top level solver algorithm is illustrated in Algorithm 1, where the functions `RESET` and `ADVANCE` are implemented in the specific solver plugin. These two functions correspond to initializing the (forward) integration at some given time, providing the necessary data, and advancing the solution to some given time point, respectively. Algorithm 1 also includes the helper function `NEXT_STEP` which checks the provided control input and determine when the next step change in the control occurs. If there are no more input step changes, the end of the simulation (N) is returned. The algorithm will ensure that the integration is stopped at such input change times. The stopping times are also used to prevent a solver plugin from taking *internal* time steps past the stopping times during the `ADVANCE` step (omitted in Algorithm 1 for simplicity).

Algorithm 1 Integration in CasADi without sensitivity analysis or events handling

```
 1: procedure SIM($x_0 \in \mathbb{R}^{n_x}$, $z_0 \in \mathbb{R}^{n_z}$, $p \in \mathbb{R}^{n_p}$, $u_k \in \mathbb{R}^{n_u}$, $k = 0,\dots,N-1$)
 2:     $k_{\text{step}} := 0$                          ▷ Index of the next input step change
 3:     for $k = 0,\dots,N-1$ do                       ▷ Forward integration
 4:         if $k = k_{\text{step}}$ then                ▷ Input step change
 5:             $k_{\text{step}} :=$ NEXT_STEP($k, u_{k+1},\dots,u_{N-1}$)
 6:             RESET($t_k, x_k, z_k, p, u_k$)
 7:         end if
 8:         $(x_{k+1}, z_{k+1}, q_{k+1}) :=$ ADVANCE($t_{k+1}$)
 9:     end for
10:     return $x_k \in \mathbb{R}^{n_x}$, $z_k \in \mathbb{R}^{n_z}$, $q_k \in \mathbb{R}^{n_q}$, $k = 1,\dots,N$
11: end procedure
```

As of this writing, there were four solver plugins available; two CasADi native fixed-step integrators implementing explicit and implicit Runge-Kutta, respectively, as well as interfaces to the SUNDIALS solvers CVODES and IDAS (Hindmarsh et al. 2005). For latter two solvers, the default algorithm is a variable-order variable-step size backward differentiation formula (BDF) method that takes successive steps *past* the given output time and then evaluates the polynomial representation available for the last integrator step at the given output time. All the interfaced solvers rely on CasADi to automatically generate any derivative information needed, including sparse Jacobians, and use sparse linear algebra for the step size computation.

The remainder of this section details how Algorithm 1 is extended internally in CasADi to be able to efficiently calculate forward and adjoint sensitivities and requires some familiarity with algorithmic differentiation. A reader mainly interested in using the framework in applications may choose to skip these parts as they are not essential for using the code.

Forward sensitivity analysis

The CasADi integrators support analytic forward sensitivity analysis via a variational approach (J. Andersson 2013; J. A. E. Andersson et al. 2019), i.e. an augmented set of DAEs are formed corresponding to the forward sensitivity equations. The forward sensitivity analysis is implemented both symbolically and numerically. In the symbolic implementation, which is the older implementation, a new DAE for the augmented system is created which is solved as any other DAE, exploiting only the sparsity of the augmented DAE system. This symbolic differentiation can be done repeatedly, to get analytic derivatives to any order, assuming sufficiently smooth DAEs.

To better exploit the specific structure of the forward sensitivity equations, a numeric implementation of forward sensitivity analysis was added (Joel Andersson 2023). The numeric implementation is implementated by supporting multiple columns in (3), corresponding to different forward seeds/sensitivities, i.e. perturbations with respect to different combinations of inputs in Algorithm 1. For N_f forward sensitivities that are calculated along with the original (undifferentiated) trajectory, the generalized definition of F can be written:

$$\begin{aligned}&\tilde{F} : \mathbb{R}^{n_x\times(1+N_f)} \times \mathbb{R}^{n_z\times(1+N_f)} \times \mathbb{R}^{n_p\times(1+N_f)} \times \mathbb{R}^{n_u\times(1+N_f)N}\\ &\to \mathbb{R}^{n_x\times(1+N_f)N} \times \mathbb{R}^{n_z\times(1+N_f)N} \times \mathbb{R}^{n_q\times(1+N_f)N}\\ &(x_0, z_0, p, u) \mapsto (x, z, q)\end{aligned} \tag{4}$$

Note that the multiple right-hand-sides are usually hidden from the user, who typically embeds the undifferentiated F from (3) in some optimization formulation, and the sensitivity equations are generated automatically to provide a gradient-based optimizer with the required derivative information.

Algorithm 1 continues to be valid when forward sensitivity equations are included in the calculation, with the only change that calculation of x_k, z_k and q_k is now done with $(1+N_f)$ columns at a time instead of one column at a time. It is up to the solver interfaces, i.e. the implementation of `RESET` and `ADVANCE` to exploit the sensitivity structure. In the SUNDIALS interfaces, this exploitation is done by providing SUNDIALS with structure-exploiting linear algebra routines. These linear algebra routines use second order derivative information – calculated via forward-over-forward algorithmic differentiation of the DAE function – to exactly and efficiently solve the augmented linear system. Note that we do not use SUNDIALS native forward sensitivity support.

Adjoint sensitivity analysis

Similar to forward sensitivity analysis, the CasADi integrators support adjoint sensitivity analysis via a variational approach (J. Andersson 2013; J. A. E. Andersson et al. 2019). These equations define a terminal-value problem coupled to the regular forward integration. Because the coupling of the terminal-value problem to the initial value problem is in one direction only, the combined problem can be solved with a forward integration, recording the integrator steps, followed by a backward integration.

The original implementation of adjoint sensitivity analysis in CasADi supported a general backward differential equation, as long as it was affine in the "backward states", and was implemented symbolically. Because of the specific structure, the integrator could be differentiated repeatedly, giving analytical sensitivities to any order, noting that adjoint-over-adjoint sensitivities can be reformulated as forward-over-adjoint sensitivities.

In CasADi 3.6, a restriction of the formulation was imposed, requiring that the terminal value problem to always be the adjoint sensitivity equations corresponding to the forward integration. The adjoint equations may in turn have forward sensitivity equations, which is important to be able to efficiently calculate second order derivative information, e.g. for numerical optimization. With N_a adjoint sensitivities and N_f forward sensitivities, (4) is further generalized as follows:

$$\begin{aligned}\hat{F} :\ & \mathbb{R}^{n_x\times(1+N_f)} \times \mathbb{R}^{n_z\times(1+N_f)} \times \mathbb{R}^{n_p\times(1+N_f)} \times \mathbb{R}^{n_u\times(1+N_f)N} \\ & \times \mathbb{R}^{n_x\times(1+N_f)N_a N} \times \mathbb{R}^{n_z\times(1+N_f)N_a N} \times \mathbb{R}^{n_q\times(1+N_f)N_a N} \\ \to\ & \mathbb{R}^{n_x\times(1+N_f)N} \times \mathbb{R}^{n_z\times(1+N_f)N} \times \mathbb{R}^{n_q\times(1+N_f)N} \\ & \mathbb{R}^{n_x\times(1+N_f)N_a} \times \mathbb{R}^{n_z\times(1+N_f)N_a} \\ & \times \mathbb{R}^{n_p\times(1+N_f)N_a} \times \mathbb{R}^{n_u\times(1+N_f)N_a N} \\ & (x_0, z_0, p, u, \lambda_x, \lambda_z, \lambda_q) \mapsto (x, z, q, \lambda_{x_0}, \lambda_{z_0}, \lambda_p, \lambda_u),\end{aligned} \tag{5}$$

where $\lambda_x, \lambda_z, \lambda_q$ correpond to *adjoint (and forward-over-adjoint) seeds* and $\lambda_{x_0}, \lambda_{z_0}, \lambda_p, \lambda_u$ correspond to *adjoint (and forward-over-adjoint) sensitivities*. Note that since z_0 is a *guess*, λ_{z_0} is going to be trivially zero, but is kept in the function signature to get a consistent function signature (that can easily be embedded into symbolic expressions). The function signature (5), which is the most complex of any of the CasADi core classes, remains the same with the addition of event support, which we will present in Section 3.

In Algorithm 2 we show the generalization of Algorithm 1 to handle forward and adjoint sensitivities, which in addition to `RESET` and `ADVANCE` mentioned earlier also includes two additional methods, `IMPULSE` to provide an additive contribution to the adjoint states at a given time and `RETREAT` to integrate the system backwards to a given time point. `NEXT_IMPULSE` is a helper function, similar to `NEXT_STEP` to find the next output time where an `IMPULSE` call is needed. Note that whenever there is a step change in a control input, the forward integration is repeated starting at the beginning of the previous step change (or initial time t_0).

As in the case of forward sensitivity analysis, the addition of numerical adjoint (and forward-over-adjoint) sensitivity analysis in CasADi 3.6 enabled significantly better structure exploitation in the integrator interfaces, specifically in the SUNDIALS interfaces. In particular, it allowed an arbitrary number of forward, adjoint and forward-over-adjoint sensitivities to be calculated along with the original simulation without increasing the size of the linear system needing to be factorized inside the interfaced ODE/DAE integrators. Similar to the forward sensitivity analysis, the forward-over-adjoint sensitivity analysis uses a matrix-free second order correction, implemented via forward-over-adjoint directional derivatives to exactly solve the augmented linear system.

Algorithm 2 Integration in CasADi with forward and adjoint sensitivity analysis but without events handling

```
1:  procedure SIM_S($x_0 \in \mathbb{R}^{n_x\times(1+N_f)}$, $z_0 \in \mathbb{R}^{n_z\times(1+N_f)}$,
        $p \in \mathbb{R}^{n_p\times(1+N_f)}$, $u_\bullet \in \mathbb{R}^{n_u\times(1+N_f)}$, $\lambda_{x_\bullet} \in \mathbb{R}^{n_x\times(1+N_f)N_a}$,
        $\lambda_{z_\bullet} \in \mathbb{R}^{n_z\times(1+N_f)N_a}$, $\lambda_{q_\bullet} \in \mathbb{R}^{n_q\times(1+N_f)N_a}$)
2:      $k_{\text{step}} := 0$      ▷ Index of the next input step change
3:      for $k = 0, \dots, N-1$ do      ▷ Forward integration
4:          if $k = k_{\text{step}}$ then      ▷ Input step change
5:              $k_{\text{prev}} := k$      ▷ Also keep track of old $k_{\text{prev}}$
6:              $k_{\text{step}} :=$ NEXT_STEP($k, u_\bullet$)
7:              RESET($t_k, x_k, z_k, p, u_k$)
8:          end if
9:          $(x_{k+1}, z_{k+1}, q_{k+1}) :=$ ADVANCE($t_{k+1}$)
10:     end for
11:     $\lambda p := 0$,   $\lambda_{u_\bullet} := 0$      ▷ Initialize to zero
12:     for $k = N-1, \dots, 0$ do      ▷ Backward integration
13:         if $k < k_{\text{prev}}$ then
14:             $k_{\text{prev}} :=$ <retrieve saved value>
15:             RESET($t_{k_{\text{prev}}}, x_{k_{\text{prev}}}, z_{k_{\text{prev}}}, p, u_k$)
16:             ADVANCE($t_{k+1}$)
17:         end if
18:         if $k < k_{\text{step}}$ then
19:             IMPULSE($\lambda_{x_{k+1}}, \lambda_{z_{k+1}}, \lambda_{q_{k+1}}$)
20:             $k_{\text{step}} :=$ NEXT_IMPULSE($k, \lambda_{x_\bullet}, \lambda_{z_\bullet}, \lambda_{q_\bullet}$)
21:         end if
22:         $[\tilde{\lambda}_x, \tilde{\lambda}_p, \tilde{\lambda}_u] :=$ RETREAT($t_k$)
23:         $\lambda_p := \lambda_p + \tilde{\lambda}_p$,   $\lambda_{u_k} := \lambda_{u_k} + \tilde{\lambda}_u$
24:     end for
25:     $\lambda_{x_0} := \tilde{\lambda}_x$
26:     $\lambda_{z_0} := 0$
27:     return $x_\bullet \in \mathbb{R}^{n_x\times(1+N_f)}$, $z_\bullet \in \mathbb{R}^{n_z\times(1+N_f)}$, $q_\bullet \in$
        $\mathbb{R}^{n_q\times(1+N_f)}$, $\lambda_{x_0} \in \mathbb{R}^{n_x\times(1+N_f)N_a}$, $\lambda_{z_0} \in \mathbb{R}^{n_z\times(1+N_f)N_a}$,
        $\lambda_p \in \mathbb{R}^{n_p\times(1+N_f)N_a}$, $\lambda_{u_\bullet} \in \mathbb{R}^{n_u\times(1+N_f)N_a}$
28: end procedure
```

3 Event support in CasADi

In order to implement event support in the CasADi integrators, we add a zero-crossing output to the DAE function (2) resulting in the generalized formulation:

$$\begin{aligned}& f : \mathbb{R} \times \mathbb{R}^{n_x} \times \mathbb{R}^{n_z} \times \mathbb{R}^{n_p} \times \mathbb{R}^{n_u} \to \mathbb{R}^{n_x} \times \mathbb{R}^{n_z} \times \mathbb{R}^{n_q} \times \mathbb{R}^{n_e} \\ & (t, x, z, p, u) \mapsto (f_{\text{ode}}, f_{\text{alg}}, f_{\text{quad}}, f_{\text{zero}})\end{aligned} \tag{6}$$

The zero-crossing component calculates n_e separate *smooth* trajectories which are monitored for zero crossings, as of this writing only from strictly negative to strictly positive values (this restriction may be removed in the future). The smoothness property is essential, and will be used for finding the exact event time as described as in Section 3.2 below. Furthermore, the smoothness property is necessary to properly calculate forward and adjoint sensitivities as described in Section 3.3 and Section 3.4, respectively.

When a zero crossing occurs, an optional *reinit* function is called. This is a separate user-provided function which has the signature:

$$\begin{aligned} &E : \mathbb{I} \times \mathbb{R} \times \mathbb{R}^{n_x} \times \mathbb{R}^{n_z} \times \mathbb{R}^{n_p} \times \mathbb{R}^{n_u} \to \mathbb{R}^{n_x} \times \mathbb{R}^{n_z} \\ &(j, t, x^-, z^-, p, u) \mapsto (x, z) \end{aligned} \tag{7}$$

where x^- and z^- are the values of x and z immediately before the event, i.e. $x^-(t) = \lim_{\tau \to t} x(\tau)$ and $z^-(t) = \lim_{\tau \to t} z(\tau)$, respectively. In other words, the function E explicitly defines a the new state vector and a new guess for the algebraic variables. If a reinit function is not provided, the identity mapping is assumed.

A differentiable function with the signature (7) can be created in various ways in CasADi. In particular, we may want to create n_e different functions of the form:

$$\begin{aligned} &E_j : \mathbb{R} \times \mathbb{R}^{n_x} \times \mathbb{R}^{n_z} \times \mathbb{R}^{n_p} \times \mathbb{R}^{n_u} \to \mathbb{R}^{n_x} \times \mathbb{R}^{n_z} \\ &(t, x^-, z^-, p, u) \mapsto (x, z), \qquad j = 0, \ldots, n_e - 1, \end{aligned} \tag{8}$$

and then use a *Switch* function in CasADi to combine them into a single function with the signature of (7). Also note that we can use an implicit definition of E or E_j e.g. by using a *Rootfinder* function in CasADi.

With the addition of the zero-crossing output in the DAE function and the new reinit function, the DAE formulation (1) becomes generalized as follows:

$$\begin{cases} (x(t), z(t)) = E(j, t, x^-(t), z^-(t), p, u(t)) \\ \qquad \text{if } \exists j : f_{\text{zero}}^{(j)}(t, x^-(t), z^-(t), p, u(t)) = 0 \\ \begin{cases} \dot{x}(t) = f_{\text{ode}}(t, x(t), z(t), p, u(t)) \\ 0 = f_{\text{alg}}(t, x(t), z(t), p, u(t)) \\ \dot{q}(t) = f_{\text{quad}}(t, x(t), z(t), p, u(t)) \end{cases} \\ \qquad \text{otherwise} \end{cases} \tag{9}$$

3.1 Generalized simulation algorithm

In Algorithm 3 we show the generalization of Algorithm 2 to also include event handling as described above. During the forward integration, the main generalization comes from allowing the `ADVANCE` step to terminate before the desired output time, in which case it will return the corresponding time and the index of the triggered root-zero crossing component. When this happens, a reinit function called `REINIT` in the algorithm is called. For simulation without sensitivities, the `REINIT` function is essentially a call to E from (7). We will show in Section 3.3 below how this function generalizes to forward sensitivity analysis. Following an event, the solver plugin needs to be reset, similarly as for the case of changing inputs. To simplify the presentation, we assume that `REINIT` returns the actual algebraic variable $\tilde{z}$. In the actual implementation, `REINIT` will just return a *guess* for the algebraic state and the actual values will be calculated during the algorithm to find consistent initial conditions inside the following `RESET`. For each event, we record x and z both before and after the event transition, along with information such as the zero crossing index and time. This will be used for the backward integration.

For the backward integration, two generalizations are necessary. Firstly, before the call to progress the backwards integration to the beginning of the interval (t_k), there is a for-loop to first visit all events that were recorded for the specific interval, in reverse order. After the adjoint integration has progressed to a specific event, the adjoint of the event transition function is called. This function is discussed in Section 3.4. Following the event, during the backward integration, we need to redo the forward integration starting at the previous event *or* input step (whichever is encountered first), denoted by the `PREVIOUS_EVENT` helper function.

3.2 Event detection algorithm

In order to determine the time of zero crossing event with high precision, the current algorithm is based on linearizing the zero-crossing algorithm in the time direction. Note that we currently do not use the zero-crossing detection capabilities of the interfaced solvers, although we may switch to doing so in a future version of the code, as discussed in Section 6.3.

Consider the zero-crossing function as a function of t, including the indirect dependencies via x, z and u:

$$e(t) = f_{\text{zero}}(t, x(t), z(t), u(t)) \tag{10}$$

We can linearize this function with respect to time as follows, assuming known values for $\dot{x}(t)$ and $\dot{z}(t)$:

$$\begin{aligned} \dot{e}(t) = &\tfrac{\partial f_{\text{zero}}}{\partial t}(t, x(t), z(t), p, u(t)) \\ &+ \tfrac{\partial f_{\text{zero}}}{\partial x}(t, x(t), z(t), p, u(t))\, \dot{x}(t) \\ &+ \tfrac{\partial f_{\text{zero}}}{\partial z}(t, x(t), z(t), p, u(t))\, \dot{z}(t), \end{aligned} \tag{11}$$

which can be efficiently calculated using a forward directional derivative of f_{zero}. Note that there are no partial derivatives w.r.t. p and u as these are constant during the interval. As of this writing, we obtain $\dot{x}(t)$ from evaluating the ODE right-hand-side, i.e. f_{ode} in (9) and did not consider zero crossing functions depending on algebraic variables. In a future iteration, we expect to obtain $\dot{x}(t)$ and $\dot{z}(t)$ from the specific integrator interface, e.g. by linearizing the DAE equations with respect to time or by evaluating an exiting polynomial representation of the $x(t)$ and $z(t)$ trajectories.

The event detection algorithm used consists of three parts:

- At the beginning of the (now generalized) `ADVANCE` function, we predict using linear extrapolation whether a zero-crossing event is expected before the given output time. If this is the case, the forward integration will be done only to this time and not to

Algorithm 3 CasADi integration, with events handling

```
 1: procedure SIM_E($x_0 \in \mathbb{R}^{n_x\times(1+N_f)}$, $z_0 \in \mathbb{R}^{n_z\times(1+N_f)}$,
        $p \in \mathbb{R}^{n_p\times(1+N_f)}$, $u_\bullet \in \mathbb{R}^{n_u\times(1+N_f)}$, $\lambda_{x_\bullet} \in \mathbb{R}^{n_x\times(1+N_f)N_a}$,
        $\lambda_{z_\bullet} \in \mathbb{R}^{n_z\times(1+N_f)N_a}$, $\lambda_{q_\bullet} \in \mathbb{R}^{n_q\times(1+N_f)N_a}$)
 2:     $k_{\text{step}} := 0$    ▷ Index of the next input step change
 3:     $t := t_0$,   $i := 0$    ▷ Current time, event index
 4:     for $k = 0,\dots,N-1$ do    ▷ Forward integration
 5:         if $k = k_{\text{step}}$ then    ▷ Input step change
 6:             $k_{\text{prev}} := k$    ▷ Also keep track of old $k_{\text{prev}}$
 7:             $k_{\text{step}} :=$ NEXT_STEP$(k, u_\bullet)$
 8:             RESET$(t_k, x_k, z_k, p, u_k)$
 9:         end if
10:         while $t < t_{k+1}$ do    ▷ Integrate until $t_{k+1}$
11:             $(\tilde{x}, \tilde{z}, \tilde{q}, t, j) :=$ ADVANCE$(t_{k+1})$
12:             while $j \geq 0$ do    ▷ Event transition(s)
13:                 Save $\tilde{x}$, $\tilde{z}$ (pre-call), $t$, $j$ for event $i$
14:                 $(\tilde{x}, \tilde{z}) :=$ REINIT$(j, t, \tilde{x}, \tilde{z}, p, u_k)$
15:                 RESET$(t, \tilde{x}, \tilde{z}, p, u_k)$
16:                 Save $\tilde{x}$, $\tilde{z}$ (post-call) for event $i$
17:                 $i := i+1$
18:                 $j :=$ <chained event, if any>
19:             end while
20:         end while
21:         $x_{k+1} := \tilde{x}$,   $z_{k+1} := \tilde{z}$,   $q_{k+1} := \tilde{q}$
22:     end for
23:     $\lambda p := 0$,   $\lambda_{u_\bullet} := 0$    ▷ Initialize to zero
24:     for $k = N-1,\dots,0$ do    ▷ Backward integration
25:         if $k < k_{\text{prev}}$ then
26:             $k_{\text{prev}} :=$ <retrieve saved value>
27:             $[t, \tilde{x}, \tilde{z}] =$ PREVIOUS_EVENT$(k, i)$
28:             RESET$(t, \tilde{x}, \tilde{z}, p, u_k)$
29:             ADVANCE$(t_{k+1})$
30:         end if
31:         if $k < k_{\text{step}}$ then
32:             IMPULSE$(\lambda_{x_{k+1}}, \lambda_{z_{k+1}}, \lambda_{q_{k+1}})$
33:             $k_{\text{step}} :=$ NEXT_IMPULSE$(k, \lambda_{x_\bullet}, \lambda_{z_\bullet}, \lambda_{q_\bullet})$
34:         end if
35:         for all events $i$ in interval $k$ in reverse order do
36:             $[\tilde{\lambda}_x, \tilde{\lambda}_p, \tilde{\lambda}_u] :=$ RETREAT$(t^{(i)})$
37:             $[\tilde{\lambda}_x, \tilde{\lambda}_z, \tilde{\lambda}_p^E, \tilde{\lambda}_u^E] :=$ ADJ_REINIT$(i, \tilde{\lambda}_x, \tilde{\lambda}_z)$
38:             $[t, \tilde{x}, \tilde{z}] =$ PREVIOUS_EVENT$(k, i)$
39:             RESET$(t, \tilde{x}, \tilde{z}, p, u_k)$
40:             ADVANCE$(t_{k+1})$
41:             $\lambda_p := \lambda_p + \tilde{\lambda}_p + \tilde{\lambda}_p^E$
42:             $\lambda_{u_k} := \lambda_{u_k} + \tilde{\lambda}_u + \tilde{\lambda}_u^E$
43:         end for
44:         $[\tilde{\lambda}_x, \tilde{\lambda}_p, \tilde{\lambda}_u] :=$ RETREAT$(t_k)$
45:         $\lambda_p := \lambda_p + \tilde{\lambda}_p$,   $\lambda_{u_k} := \lambda_{u_k} + \tilde{\lambda}_u$
46:     end for
47:     $\lambda_{x_0} := \tilde{\lambda}_x$
48:     $\lambda_{z_0} := 0$
49:     return $x_\bullet \in \mathbb{R}^{n_x\times(1+N_f)}$, $z_\bullet \in \mathbb{R}^{n_z\times(1+N_f)}$, $q_\bullet \in$
        $\mathbb{R}^{n_q\times(1+N_f)}$, $\lambda_{x_0} \in \mathbb{R}^{n_x\times(1+N_f)N_a}$, $\lambda_{z_0} \in \mathbb{R}^{n_z\times(1+N_f)N_a}$,
        $\lambda_p \in \mathbb{R}^{n_p\times(1+N_f)N_a}$, $\lambda_{u_\bullet} \in \mathbb{R}^{n_u\times(1+N_f)N_a}$
50: end procedure
```

the output time. If there are multiple zero crossing events predicted, only the soonest one will be considered. Also, ommitted in the algorithm for ease of presentation, if a zero-crossing event is predicted before the next input change, the stopping time for internal time stepping will be updated accordingly.

- If after this initial integration, the zero crossing functions and their derivatives w.r.t. time indicate that a zero crossing event has occurred or is still predicted to occur before the desired output time, a root-finding iteration will start. The algorithm is an Newton method, with a fallback to bisection if $\dot{e}$ has the wrong sign. This fallback can e.g. happen if $\dot{e}_j$ is non-positive, even though the sign of e_j indicates that a zero crossing from negative-to-positive has occured, or if the predicted event crossing happens before the start of the integration interval. During the rootfinding iterations, the solver interfaces will be responsible for updating the state to a given time (which may require small steps backwards in time).

- When the zero crossing iteration has reached a given tolerance, or hit a user-selected maximum number of iterations, the corresponding values for x, z and q along with time and zero-crossing index are returned to the user.

We do not include specific handling of the case where the event time is explicitly given, e.g. as a function of p, u and non-changing components of x, but note that the above algorithm will find the exact time of such events in a single iteration since $e(t)$ is linear in t.

3.3 Forward sensitivity analysis

For the forward sensitivity analysis, the function `REINIT` in Algorithm 3 needs to be generalized. To get the correct sensitivity propagation through the event, we must take into consideration that the event time t may depend on the state. We can handle this at the event considering the time t to be implicitly defined by the corresponding zero crossing function:

$$f_{\text{zero}}^{(j)}(t, x, z, p, u_k) = 0 \quad \Leftrightarrow \quad t = G(x, z, p, u_k) \tag{12}$$

We can propagate forward sensitivities through this function using the implicit function theorem, similar to how forward sensitivities for CasADi's `Rootfinder` class implemented. Since it is a scalar function, the propagation can easily be calculated:

$$\begin{aligned}\hat{t} &:= \frac{\partial t}{\partial x}\hat{x} + \frac{\partial t}{\partial z}\hat{z} + \frac{\partial t}{\partial p}\hat{p} + \frac{\partial t}{\partial u}\hat{u} \\ &= -\frac{1}{\dot{e}_j}\left(\frac{\partial f_{\text{zero}}}{\partial x}\hat{x} + \frac{\partial f_{\text{zero}}}{\partial z}\hat{z} + \frac{\partial f_{\text{zero}}}{\partial p}\hat{p} + \frac{\partial f_{\text{zero}}}{\partial u}\hat{u},\right)\end{aligned} \tag{13}$$

where $\hat{t}$ are the forward sensitivities of t and the corresponding forward seeds are $\hat{x}$, $\hat{z}$, $\hat{p}$ and $\hat{u}$, respectively.

With $\hat{t}$ for each sensitivity direction calculated, we are able to propagate the forward sensitivities through the reinit function:

$$\hat{x}_E := \frac{\partial E_x}{\partial t}\hat{t} + \frac{\partial E_x}{\partial x}\hat{x} + \frac{\partial E_x}{\partial z}\hat{z} + \frac{\partial E_x}{\partial p}\hat{p} + \frac{\partial E_x}{\partial u}\hat{u}, \quad (14)$$

where E_x is the calculation of x using E in (7). This calculation is performed using a forward directional derivative applied to the reinit function (7). Since the reinit function will only provide a guess for z (the exact value being determined by the DAE), no derivative propagation is needed.

Finally, we need to consider that sensitivity of $\hat{t}$ needs to be propagated to the duration of the subsequent interval. For example, if a small perturbation Δp in an input parameter p leads to the event happening a time Δt later, the subsequent integration interval will be Δt shorter. We can account for this by using $\dot{x}$ obtained from (9) and the known sensitivity in duration length $(-\hat{t})$.

$$\hat{x}_{\text{REINIT}} := \hat{x}_E - \hat{t}\,\dot{x} \quad (15)$$

3.4 Adjoint and forward-over-adjoint sensitivity analysis

Algorithm 3 also include a tentative implementation of adjoint sensitivity analysis and second order (forward-over-adjoint) sensitivity analysis. During the backward integration, there is no need to detect zero crossings. Instead we will simply keep records of the events (times and corresponding event indices) during the forward integration and then visit the same events in reverse order during the backward integration. Second order derivatives are handled by allowing all variables to have multiple right-hand-sides.

As of this writing, the extension of the adjoint sensitivity support to support events is still ongoing. The parts in Algorithm 3 related to adjoint and forward-over-adjoint sensitivity analysis therefore reflects the planned implementation.

4 Examples

4.1 Forward sensitivities for a bouncing ball

In our first example, we perform an analytical forward sensitivity analysis for a bouncing ball and compare the results with a finite-difference approximation. The system has two states corresponding to height h and velocity v, i.e. the state vector is $x = [h; v]$. The corresponding ODE is:

$$\dot{h} = v, \quad \dot{v} = -9.81 \quad (16)$$

When the ball hits the ground at $h = 0$, defined by $f_{\text{zero}}(x) = -h$, an event will be triggered defined by:

$$x = \begin{bmatrix} 0 \\ -0.8\,v^-, \end{bmatrix} \quad (17)$$

where v^- is the velocity immediately before the event.

In the leftmost figures of Figure 1, we show the event simulation, over 7 s for a ball starting at rest at $h = 5$, using SUNDIALS/CVODES as the interfaced solver. The remaining figures show the sensitivities of h and v with respect to perturbations in $h(0) = h_0$ and $v(0) = v_0$, respectively. The results are compared to a basic finite differencing perturbation of the whole simulation trajectory.

To understand the results in the lower right subplot, which may seem counter-intutitive, it can be shown that for a ball starting at rest, the derivative of the time of the first bounce T_{bounce} with respect to initial velocity can be written:

$$\frac{dT_{\text{bounce}}}{dv_0} = \frac{1}{g}. \quad (18)$$

Therefore, the first derivative of the ball velocity at impact $v_{\text{impact}} = v_0 - g\,T_{\text{bounce}}$ with respect to initial velocity is zero:

$$\frac{\partial v_{\text{impact}}}{\partial v_0} = \frac{dv_0}{dv_0} - g\frac{dT_{\text{bounce}}}{dv_0} = 1 - \frac{g}{g} = 0. \quad (19)$$

The first order sensitivity of the ball velocity *after* the bounce with respect to initial velocity, is therefore just due to how much time has elapsed since the bounce:

$$\frac{dv(t; h_0, v_0)}{dv_0} = -\frac{dT_{\text{bounce}}}{dv_0}(-g) = 1. \quad (20)$$

This theoretical result, which holds in the *almost everywhere* sense, is confirmed with the analytical forward sensitivities (blue line). The result repeats itself at subsequent bounces. For the corresponding finite difference approximation (red line), in contrast, the numerical error will grow for every bounce.

4.2 Parameter estimation for a breaking spring

As a second example, we consider the a simple model of a spring formulated in Modelica. When the spring is extended too far, an event corresponding to the spring "breaking" is triggered:

```
model BreakingSpring
    input Real m(start = 1)
      "PARAMETER:Mass";
    output Real v(start = -5, fixed = true)
      "velocity";
    output Real x(start = -1, fixed = true)
      "displacement";
    input Real k(start = 2)
      "PARAMETER:spring constant";
    input Real c(start = 0.1)
      "PARAMETER:damping constant";
    input Real d(start = 0) "disturbance";
    Real f "spring force";
    Boolean b "Is the spring broken?";
initial equation
  b  = false;
equation
    der(x) = v;
    f = if not b then -k * x + d else 0;
```

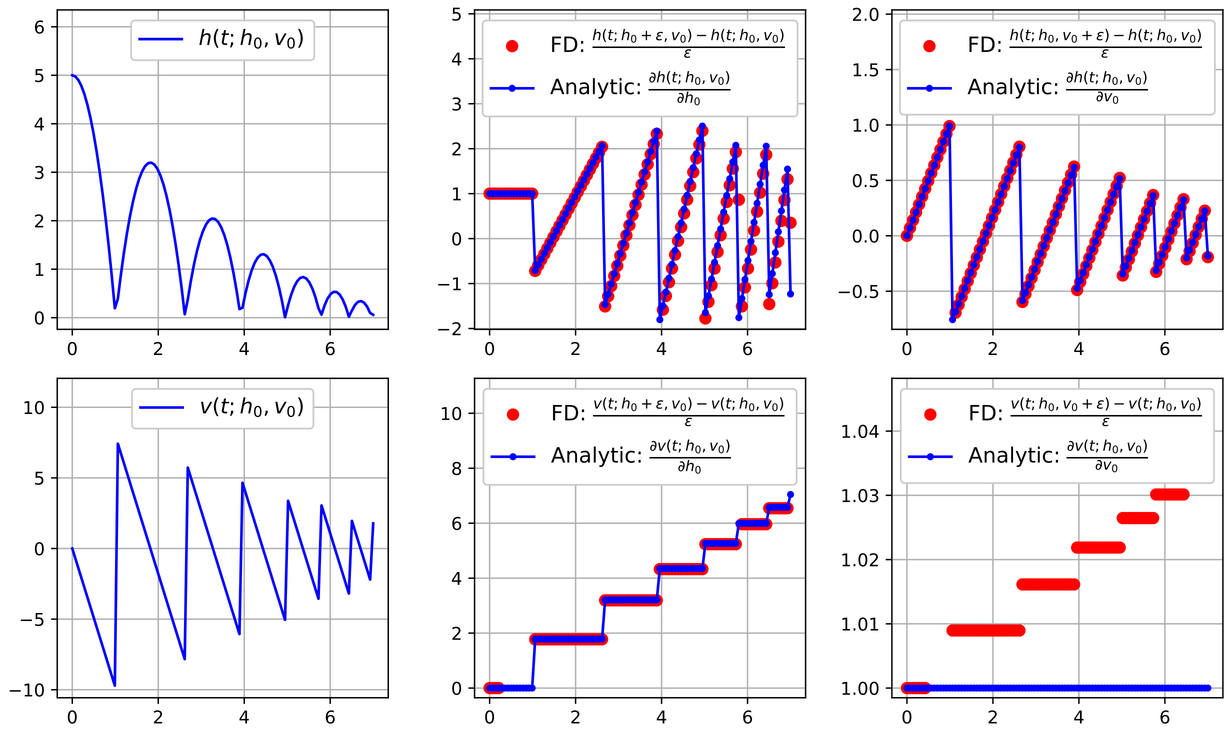

Figure 1. Forward sensitivity analysis for a bouncing ball, comparison with finite differences (FD)

```
    m * der(v) + c * v = f;
    when x>2 then
      b = true;
    end when;
end BreakingSpring;
```

Compared to the bouncing ball model, the breaking spring model includes the following:

- A free input parameter `d`, corresponding to u in (1)
- Three tunable parameters, `m`, `k` and `c`, corresponding to p in (1). To ensure derivative information is available after compiling the model (e.g. into an FMU), we will model tunable parameters as *controls*, using a `Parameter:` prefix in the description string to distinguish them from regular controls.
- A boolean state `b`, which is updated discontinuously at events. Since CasADi does not have the concept of discrete states, we will model discrete states as real-valued states with zero time derivative, i.e. `b` is a component of x (say, index i_b) with $\dot{x}[i_\text{b}] = 0$.

Using OpenModelica 1.24, we compile the above model into an XML file, containing a symbolic representation of the problem, using the approach described in (Shitahun et al. 2013). This model is then imported into a CasADi `DaeBuilder` instance, which in turn is used to generate an analytically differentiable integrator object in CasADi, again using SUNDIALS/CVODES as the interfaced solver.

CasADi integrator instances can be embedded into expression graphs corresponding to different optimization formulations. In Figure 2 (left), we show the result of solving a parameter estimation problem using the hybrid integrator. The problem corresponds to finding the parameter values m, k and c that minimize a sum-of-squares cost function:

$$\underset{m,k,c}{\text{minimize}} \quad \sum_{k=1}^{N} (x_k - \tilde{x}_k)^2, \tag{21}$$

subject to the hybrid dynamical equation and bounds of the parameter. To generate simulated measurements $\tilde{x}_k$, we add Gaussian noise to the simulation result corresponding to known values of the parameters. The optimization is done for a known disturbance vector d, but again with random noise added, as shown in Figure 2 (right). The problem is solved using a single-shooting discretization, using IPOPT as an optimizer.

5 Summary

In this work, we have shown an extension of the DAE simulation routines in CasADi to handle systems with events. This includes the efficient calculation of analytical sensitivity information, as needed by gradient-based optimization algorithms, also in the presence of events. We provided details of the forward sensitivity implementation, illustrated with two examples, as well as details on the ongoing work to implement adjoint and forward-over-adjoint sensitivity analysis with events. While we have thus far relied on relatively simple toy examples available as CasADi

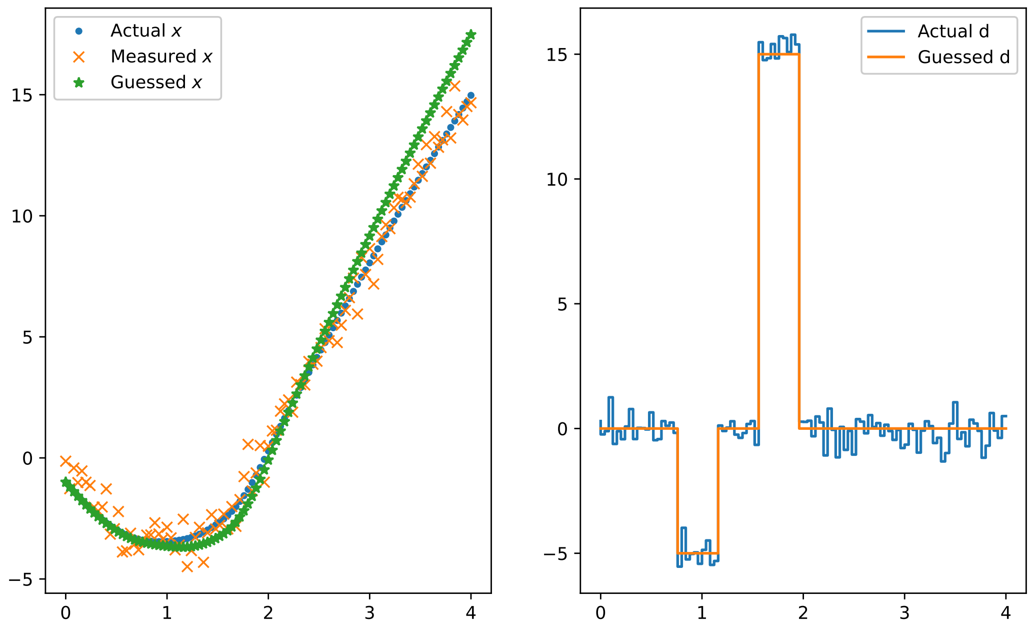

Figure 2. Parameteter estimation for a breaking spring, with generated measurement values.

symbolic expression graphs, the intention is to use this feature to implement dynamic optimization for challenging cyber-physical systems, including but not limited to systems implemented in Modelica. We will discuss the path to handle such systems in the following section.

6 Outlook

The work presented in this paper is in active development, with additional features being added as they become required by applications. In the following, we discuss some of the most important extensions planned.

6.1 Event support for models provided as standard FMUs

The ultimate goal of this work is to enable the formulation and solution real-world optimization problems with event dynamics, in particular those formulated in Modelica. In our initial experiments, presented in Section 4, we used a symbolic coupling based on a legacy XML-based symbolic coupling between OpenModelica and CasADi. This coupling is neither well maintained, nor generic enough to handle realistic systems. It is also restricted to a single exporting tool (OpenModelica).

A recent addition to CasADi is the import of general FMUs adhering to FMI 2, as described in (Joel Andersson 2023). In pre-release versions of CasADi, this support has since been extended to FMI 3, including the interface to adjoint derivatives of model equations. Our plan is to use the FMI interface together with the event support in the CasADi integrators to be able to efficiently and conveniently solve optimization problems for real-world Modelica models. Note that by relying on FMI, the structure of the underlying Modelica model becomes irrelivant as long as it conforms with the FMI standard and has the prerequisite smoothness properties for numerical optimization. It is also possible to use models that include variables that cannot be represented in CasADi, for example records or string-valued expressions, as long as these variables are not manipulated by the optimizer.

Since the FMI format, as written, does not natively conform to the required formulation (9), some reformulations of the Modelica models may be needed prior to FMU generation. In particular:

- Event indicator expressions will need to be linked to differentiable model outputs. That means that the argument of *when*-constructs in Modelica may need to be assigned to additional model outputs, following some naming convention. This convention ensures that derivative information is available for the zero-crossing functions.
- The reinit equations need to consist of simple *outputs-to-states* mappings. This means that at events, the differential state should be assigned to some of variable with output causality. Each event indicator should uniquely map to an assignment, which may require the addition of additional output variables. This convention ensures that derivative in-

formation is available for the reinit functions.

- We may need to reformulate free parameters as inputs (as in Section 4.2) to ensure that analytic derivative information with respect to these parameters is included in the FMU. Alternatively, we can rely on tool-specific extensions, such as using the annotation `"evaluate = false"` in Dymola to ensure that the parameter can be manipulated by the optimizer.

6.2 A standardized symbolic interface based on *Base Modelica*

A symbolic model interface, such as the XML-based interface used in Section 4.2 will always have some fundamental advantages over a "black box" binary interface. This is especially true when the model dimensions are small or when higher order derivative information is needed. To be able to take advantage of the fundamental advantages of a symbolic interface, we plan to replace the XML interface with a new symbolic interface based on a ANTLR4-based parser for the emerging *Base Modelica* standard (Kurzbach et al. 2023). This interface builds on our previous work with Pymoca and Cymoca, cf. `https://github.com/pymoca/pymoca` and `https://github.com/jgoppert/cymoca`, respectively.

Since Base Modelica is intended to become a standard, with ongoing work to export models in this format from different Modelica compilers, the approach should be compatible with multiple tools. The hope is also that since Base Modelica is in essence a small subset of the full Modelica language, implementing and maintaining a parser should be possible with a reasonable effort.

6.3 Event detection in interfaces

In Section 3.2, we presented an approach to locate events based on an algorithm implemented in the integrator base class. An alternative to this algorithm would be to use the solver's native event-finding algorithm, such as the Illinois algorithm (Hiebert and Shampine 1980) used in SUNDIALS. This algorithm has proven efficient and robust for numerous applications. There is also value in using the same event finding algorithm as the modeler uses for hybrid simulation.

6.4 Algebraic variables in the zero-crossing functions and reinit functions

The implementation of the proposed approach was done in a way that was generic for both ODEs and DAEs, although it had yet to be tested with DAEs as of this writing. The implementation would also need an extension to be able to handle the case when algebraic variables (z) explicitly enter in the zero-crossing functions or reinit functions.

Disclaimer

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

Acknowledgements

This material is based on research sponsored by DARPA under agreement number FA8750-24-2-0500. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

The authors also want to acknowledge the helpful remarks by the anonymous reviewers.

References

Andersson, J. (2013-10). "A General-Purpose Software Framework for Dynamic Optimization". PhD thesis. Arenberg Doctoral School, KU Leuven.

Andersson, Joel (2023-10). "Import and Export of Functional Mockup Units in CasADi". In: *Proceedings of the 15th International Modelica Conference*. Vol. 2855, pp. 321–326.

Andersson, Joel A E et al. (2019). "CasADi – A software framework for nonlinear optimization and optimal control". In: *Mathematical Programming Computation* 11.1, pp. 1–36. DOI: 10.1007/s12532-018-0139-4.

Corner, Sebastien, Corina Sandu, and Adrian Sandu (2019). "Modeling and sensitivity analysis methodology for hybrid dynamical system". In: *Nonlinear Analysis: Hybrid Systems* 31, pp. 19–40. ISSN: 1751-570X. DOI: https://doi.org/10.1016/j.nahs.2018.07.003. URL: https://www.sciencedirect.com/science/article/pii/S1751570X1830058X.

Hiebert, Kathie L. and Lawrence F. Shampine (1980-02). *Report SAND80-0180: Implicitly Defined Output Points for Solutions of ODEs*. Tech. rep. Sandia National Laboratory.

Hindmarsh, Alan C et al. (2005). "SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers". In: *ACM Transactions on Mathematical Software (TOMS)* 31.3, pp. 363–396. DOI: 10.1145/1089014.1089020.

Kurzbach, Gerd et al. (2023-10). "Design proposal of a standardized Base Modelica language". In: *Proceedings of the 15th International Modelica Conference*. Vol. 2855, pp. 469–478.

Modelica Association (2021-02). *Modelica – A Unified Object-Oriented Language for Systems Modeling. Language Specification Version 3.5*. Tech. rep. Linköping: Modelica Association. URL: https://specification.modelica.org/maint/3.5/MLS.html.

Rackauckas, Christopher and Qing Nie (2017). "DifferentialEquations.jl–a performant and feature-rich ecosystem for solving differential equations in Julia". In: *Journal of Open Research Software* 5.1.

Shitahun, Alachew et al. (2013). "Model-Based Dynamic Optimization with OpenModelica and CasADi". In: *IFAC Proceedings Volumes* 46.21. 7th IFAC Symposium on Advances in Automotive Control, pp. 446–451. ISSN: 1474-6670. DOI: https://doi.org/10.3182/20130904-4-JP-2042.00166. URL: https://www.sciencedirect.com/science/article/pii/S1474667016384117.